

**TRANSLATION OPTIMIZATION AND PARALLELIZATION OF  
GENETIC ALGORITHMS  
A DISCOURSE OF IMPLEMENTATION INTO C++**

A Senior Scholars Thesis

by

AVERY ADAM WHITE

Submitted to the Office of Undergraduate Research  
Texas A&M University  
in partial fulfillment of the requirements for the designation as

UNDERGRADUATE RESEARCH SCHOLAR

April 2010

Major: Computer Engineering

**TRANSLATION OPTIMIZATION AND PARALLELIZATION OF  
GENETIC ALGORITHMS  
A DISCOURSE OF IMPLEMENTATION INTO C++**

A Senior Scholars Thesis

by

AVERY ADAM WHITE

Submitted to the Office of Undergraduate Research  
Texas A&M University  
in partial fulfillment of the requirements for the designation as

UNDERGRADUATE RESEARCH SCHOLAR

Approved by:

Co-Research Advisors:

Associate Dean for Undergraduate Research:

Emily Zechman  
Alex Sprintson  
Robert C. Webb

April 2010

Major: Computer Engineering

## ABSTRACT

Translation, Optimization, and Parallelization of Genetic Algorithms  
A Discourse of Implementation into C++. (April 2010)

Avery Adam White  
Department of Electrical and Computer Engineering  
Texas A&M University

Co-Research Advisors: Dr. Emily Zechman  
Dr. Alex Sprintson

Department of Civil Engineering  
Department of Electrical and Computer Engineering

A genetic algorithm approach can be used to address complex management and design problems within the discipline of civil engineering, such as managing storm water from urban areas, which can degrade environmental conditions. Additionally, conventional programming practices often produce serial code unfit for taking full advantage of a multi-core architecture. This research examines the methodology behind implementing a simulation-optimization approach in a serial versus parallel environment. In Chapter I, a brief history of the project is presented as well as the motivation for the research. In Chapter II, the concepts behind developing an optimized serial genetic algorithm are examined. In Chapter III, a parallel design methodology is implemented. In Chapter IV, the serial and parallel results are examined. It is from these results that a speedup of 1.61 can be observed if OpenMP parameters are finely tuned. It can also be observed that without careful selection of subroutines and the exploitation of their inherent parallelism,

runtime overhead can significantly degrade performance beyond even the serial implementation. As such, optimization of OpenMP parameters is recommended. In Chapter V, and concluding chapter, a method formulated from the research is presented that addresses how to develop an optimized and parallelized genetic algorithm that targets complex civil engineering management and design problems.

## DEDICATION

*Soon exposed all liberty*

*To reason out dichotomy*

*Once separate and now made free*

*Commensurate and pleasantry*

*No creature small or large can touch*

*The realm of Man who thinks on such*

*Matters grown divide and much*

*Logic serves a many crutch*

*Doth now consider time well spent*

*Or wasted end and will repent*

*What of the gain to had in life?*

*For in this world are riches rife*

*But all for naught once on the pyre*

*Are fruitless ends of mans d'sire*

*So pray I've spent time graciously*

*In thanking Him who once made me*

*I don't deserve so great a grace*

*But promised I to see His face*

*And document to celebrate*

*And to my Maker, dedicate*

## ACKNOWLEDGEMENTS

Let the reader know that had it not been for the outrageous kindness of Dr. Emily Zechman, this would not be at all possible. It is with sincerest gratitude that I give her the prime position as first on this list. Her unmatched patience, wonderfully immersive teaching style, and faith in her students has given me the quiet confidence that I have needed for some time. Some teachers tell you about the bridge, and draw diagrams to show you how to cross it. Dr. Zechman has walked it over and back taking students like me with her each and every time.

Secondly, I would not have remained in the department of engineering had it not been for the unfailing support of my parents. Parents who have supported me spiritually, mentally, emotionally, and financially; believing in me when I didn't believe in myself, they have blessed me beyond all reason, and I may never know fully how blessed I am to call them my own. I love them dearly.

Third, I would like to extend a warm thanks to Dr. Alex Sprintson, who, by the faith of his colleagues, agreed to take me on as an undergraduate research student and has continually extended his patience and willingness to help me complete this endeavor.

Fourth, I would like to thank Daniel Eng and Darel Booth, who both helped me learn the language of C++, programming practices, standards, and above all, how to begin to

travel down the road in thinking like a computer scientist. They are both wonderful friends and knowledgeable in their fields.

Finally, I would like to acknowledge the Undergraduate Research Scholars program, and Dr. Suma Datta. The Department of Electrical and Computer Engineering, and Dr. Costas Georgiades, Dr. Karen Butler, and Dr. John Tyler.

## TABLE OF CONTENTS

	Page
ABSTRACT.....	iii
DEDICATION.....	v
ACKNOWLEDGEMENTS.....	vi
TABLE OF CONTENTS.....	viii
LIST OF FIGURES.....	x
LIST OF CODE FRAGMENTS.....	xi
LIST OF EQUATIONS.....	xii
LIST OF TABLES.....	xiii
 CHAPTER	
I      INTRODUCTION.....	1
Applications of parallel genetic algorithms.....	2
II     METHODOLOGY AND DEVELOPMENT OF A SERIAL GENETIC ALGORITHM IN C++.....	5
The genetic algorithm.....	5
Genetic terminology and the GA.....	5
Genetic algorithm operators.....	8
Operator order: the definition of a genetic algorithm.....	11
A complete GA.....	12
Testing serial GA performance for the Rosenbrock function.....	12
III    METHODOLOGY AND DEVELOPMENT OF A PARALLEL GENETIC ALGORITHM IN C++.....	17
Parallelization methodology and the OpenMP API.....	17
The feature set.....	21



CHAPTER	Page
Environment variables.....	24
Environment subroutines.....	25
Parallelization approach for a genetic algorithm.....	27
IV SERIAL RESULTS.....	29
V PARALLEL RESULTS.....	38
Optimization.....	41
VI CONCLUSIONS.....	44
REFERENCES.....	46
APPENDIX A.....	49
CONTACT INFORMATION.....	82

## LIST OF FIGURES

FIGURE	Page
1      Best fitness for 1000 generations.....	14
2      Average fitness for 1000 generations.....	15
3      Fork and join model.....	18
4      Average fitness for 250 generations.....	30
5      Average fitness for 500 generations.....	31
6      Average fitness for 750 generations.....	32
7      Best fitness for 250 generations.....	33
8      Best fitness for 500 generations.....	34
9      Best fitness for 750 generations.....	35

## LIST OF CODE FRAGMENTS

FRAGMENT	Page
1      Example illustrating parallelizable for loop.....	20
2      Example illustrating destroyed parallelism.....	20
3      Parallelized initialization operator.....	38
4      Parallelized evaluate operator.....	39
5      Parallelized selection operator.....	40
6      Parallelized crossover operator.....	40
7      Parallelized generation switch operator.....	41

## LIST OF EQUATIONS

EQUATION		Page
1	Rosenbrock's Function.....	12
2	Equation for calculating speedup.....	42

**LIST OF TABLES**

TABLE		Page
1	GA parameter settings.....	16
2	Serial performance with different parameter settings.....	36
3	Runtimes and OpenMP parameters for parallel implementation.....	43

## CHAPTER I

### INTRODUCTION

Evolutionary computation (EC) methods are increasingly used to solve the complex, multi-objective, and ill-posed optimization models that arise in engineering design and management problems. EC methods, such as a genetic algorithm, use a population-based search and execute a set of operators to converge to a near-global optima.

Application of EC-based methodologies for engineering problems typically requires a simulation-optimization framework, which couples an optimization method with a simulation model, to iteratively guide the search. Examples of simulation-optimization frameworks for engineering applications include groundwater source identification (Mirghani et al., 2009), urban water supply headworks optimization (Kuczera 1992), and allocation of chlorination stations in a water distribution network (Ewald et al., 2008).

One limitation of applying EC-based methods for engineering problems is the simulation time that these engineering system models require. Heuristic optimization methods typically require tens of thousands of simulation executions, resulting in an impracticable computation time. Parallel implementation of EC-based methods can take advantage of distributed computing capabilities by distributing computations across

---

This thesis follows the format of *Journal of Water Resources and Planning Management*.

several processors.

Parallelization utilizes either a standalone machine with several 'cores' embedded within its microarchitecture, or a 'cluster' of machines harnessed together to form a single computational unit. The clock speed of each 'core' (raw processing element) contained within a microprocessor determines the overall capabilities of the chip, and therefore, the machine. Parallel genetic algorithms (PGAs) have been found to offer significant speedup over their serial counterparts, reducing runtime by including more processors (Cantu-Paz and Goldberg, 1999; Battiti and Tecchiolli, 1992). Since the introduction of the dual core processor by IBM in 2001 (Tendler et al., 2002), the existing work in PGAs (e.g. Muhlenbein et al., 1991) has been further developed to solve more challenging applications, including a set of engineering applications (Mirghani et al., 2009; Kuczera 1992; Ewald et al., 2008). These examples, however, remain as specific implementations and an algorithm design methodology remains undocumented. As a result, the best practices for GA parallelization techniques as applied to solve engineering problems is a largely untapped research topic.

### **Applications of parallel genetic algorithms**

The PGA has been applied to several engineering domains to take advantage of the distribution of calculations across a small cluster, in fields ranging from genetics and genome scanning (Rausch et al., 2008), construction projects (Kandil and El-Rayes 2006), and even research pertaining to hard drive disk bearings (Wang, Tsai, and Cha 2009). In the hard drive disk bearings study, Wang, Tsai, and Cha (2009) utilized a

Fortran implementation of Cluster OpenMP, an application programming interface (API) with symmetric multiprocessing support (SMP) specification developed by Intel.

Further, it was found in implementing the OpenMP model that resultant code exhibited good speedup and scalability for inverse problems, while maintaining its portability across shared memory parallel systems (Wang and Wu, 2002). At the compilation stage, when programs are 'built', the OpenMP specification dictates to the compiler how to 'parallelize' the code. The groundwater study by Mirghani et al. (2009) utilized a PGA, based on the Message Passing Interface (MPI) communications protocol, in which parallelization occurs at runtime. MPI is typically selected as a distributed computing protocol based on its optimal run-times, but this implementation may not be as reproducible, due to the demands placed on the programmer to understand the underlying principles of shared memory management. As Wang et al. discovered, OpenMP may be a better choice given implementation limitations. In addition, OpenMP has demonstrated excellent speedup, achieving a 450% speedup over a serial implementation of a least squares equation solving portion of an algorithm, and a 240% speedup overall (Wang and Wu, 2002). Also, according to the study performed by Kegel et al. (2009), the relatively easily implementable OpenMP directive set slightly outperformed even the explicit Thread Building Blocks (TBB) developed by Intel ®.

The purpose of this research is to develop a PGA based on the OpenMP specification. To provide further guidance for future researchers, design methodologies will be identified for implementing a PGA that can be applied for engineering problems.



Included is a discussion of how parallelized code development is augmented from serial-code development practices during the pseudo-code stages of the software life-cycle.

Results are included to demonstrate the speedup of the PGA over a standard GA for a test problem.

## **CHAPTER II**

### **METHODOLOGY AND DEVELOPMENT OF A SERIAL GENETIC ALGORITHM IN C++**

#### **The genetic algorithm**

“Living organisms are consummate problem solvers”, states John Holland in his 1992 composition, a reference to genetic algorithms and their heavy dependence on the theory of natural selection (Holland 1992). The underlying concept of a genetic algorithm (GA) is that a population, mimicking those found in nature, evolves towards the identification of a “best” single individual using a predetermined set of rules. While traditional search methods often involve Hill-Climbing methods, a GA stochastically produces a population and evolves the individuals through genetic operators. At the core, the simplest GA only needs the operators of selection, crossover, and mutation, each either manipulating the population directly or indirectly (Mitchell 1996). Utilizing this search regimen, the problem of only finding local optima is reduced as the entire solution space is explored. This contrasts with Hill-Climbing methods, which identifies a global solution only when a potential solution is seeded proximal to the global minima.

#### **Genetic terminology and the GA**

Due to the GA’s extensive emulation of nature, biological terms have been borrowed and deserve cursory explanation. Accordingly, they are contrasted by demonstrating

their role both in nature, as well as the algorithm.

### *Population*

In nature, a population represents a collection of living organism of varying types.

Within the algorithm, the population is a set of individuals that capture a limited portion of the solution space at each generation.

### *Individual*

In nature, an individual represents a single unit belonging to a population. With respect to the algorithm, this is a single encoded solution to the given problem. It is divided into, or consists, entirely of, chromosomes.

### *Chromosomes*

In nature, chromosomes represent the genetic blueprint of the organism contained with strings of DNA. In a GA, a chromosome is the term for the collection of decision variables, or genes.

### *Genes*

In nature, each gene is responsible for determining a specific trait of the individual. In a GA, a gene is a specific decision variable. The complexity of a problem usually corresponds to the number of decision variables involved.

### *Alleles*

In nature, an allele is a collection of possible DNA sequences that any given gene may exhibit; these are the potential 'settings' for a trait (Mitchell 1996). With respect to a GA, an allele is the collection of possible values a decision variable can have. These may be a set of integer or binary numbers or a range of real numbers.

### *Fitness*

In nature, the fitness of an individual is subjectively defined as its probability of survival. This is much more concrete within the context of a GA in that the decision variables are fed through a 'rule' or "Fitness Function" which calculates the fitness numerically. It is this property that ultimately determines the individual's role within the context of the Operators, as described below.

## **Genetic algorithm operators**

### *Initialization*

Before the algorithm is ever run, a starting population must be created and seeded with random values for every decision variable. This can be implemented with a simple constructor in most computer languages.

### *Evaluation*

While sometimes contained within other operators, evaluation is the process of taking the decision variables and passing them through the fitness function to generate the individual's fitness. Depending on the implementation, this operator may be integrated into other operators, or run several times throughout the algorithm.

### *Selection*

This operator is chiefly responsible for running a simple search algorithm to select the individuals that will reproduce. The individual's fitness plays a role in determining its reproductive frequency. Within this particular implementation, tournament selection is utilized. This method compares the fitness of two randomly selected individuals and selects the one with the higher fitness.

### *Crossover*

As mentioned above, this is the computational equivalent to procreation. However, a key difference here is that individual crossings are subject to a 'crossover rate' as determined

by the implementation. If two individuals are crossed (e.g. Parent A and Parent B), then the chromosomes of one are switched with the chromosomes of the other at an implementation defined 'crossover point', or locus in which all the genes before or after the crossover point are switched from Parent A to Parent B.

As a simple example, consider two parents (A and B) each with two genes (0 and 1) contained within their respective chromosomes. The algorithm proceeds as follows:

- 1) Save both Parents.
- 2) Create Child A by taking Parent B's gene 1 and placing it at its gene 0 locus, and Parent A's gene 1 placing it at its gene 1 locus.
- 3) Create Child B by taking Parent A's gene 1 and placing it at its gene 0 locus, and Parent B's gene 1 and placing it at its gene 1 locus.

Traditionally, the children are stochastically reinserted into the population. Crossover can be implemented any one of many available ways. Other possibilities such as non-deterministic gene selection uses a rule for crossover that involves random selection of genes, and recombining two individuals may yield different results with each iteration. GAs provide excellent headroom for creativity, which simultaneously allows for greater adaptability to problems with increasing complexity.

### *Mutation*

Following the seemingly random patterns of nature, DNA patterns rearrange for no apparent reason. Sometimes, the results are catastrophic and increase the fatalities within the species, however, mutation can better equip organisms within a species and therefore increase their survivability within the population. This operator brings in slightly more random chance through the probability of mutation (often much lower than the probability of crossover), which may result in an individual with a higher fitness. Mutation randomly selects a gene and changes its value to a new random allele.

### *Elitism*

In a simple implementation, elitism saves the most fit individual contained in a population before any other operators run, and compares it to the individual with the best resultant fitness after the execution of all other operators. The best solution is then re-inserted into the population if the solution quality has been degraded through execution of other operators. Elitism ensures that the best solution will not be lost over generations.

### *Generation switch*

An operator native to the particular implementation being discussed here, Generation Switch is responsible for copying the next generation (i.e. what the algorithm *has* worked on) into the current generation. This is done to advance the algorithm from one

generation to the next.

### **Operator order: the definition of a genetic algorithm**

It is the following specific combination of operators within a GA that distinguishes it from other search methods (Mitchell 1996):

- 1) Parallel\* population-based search with stochastic selection of many individuals.
- 2) Stochastic crossover
- 3) Stochastic mutation.

“Parallel” in this context is parallel in regard to the search methodology and should not to be confused with “parallel programming”.



## A complete GA

### 1) Initialization

For loop that runs over the number of desired generations:

2) Evaluate

3) Elitism - Save

4) Selection

5) Crossover

6) Mutation

7) Elitism – Compare – Take Action if necessary

> Move the next population to the current population to ready the algorithm for the next run

> Return to the top of the 'for' loop

When for loop terminates output the best resultant solution.

## Testing serial GA performance for the Rosenbrock function

The simple Rosenbrock's Valley function is used to demonstrate the serial implementation of the GA. The Rosenbrock function is multi-modal, and local minima are relatively easy to find, while the global minima is inherently more difficult:

$$f(x,y) = (1 - x)^2 + 100 * (y - x^2)^2$$

Equation 1. *Rosenbrock's function*

The optimal solution (1,1) is known, and this problem lends itself to a GA with two decision variables (x and y). Solutions with fitness values close to zero are near to the global minimum.

The convergence of the GA to the global minimum can be seen in Figs. 1 and 2, which demonstrate the convergence of the best individual and the average fitness value of the population over 1000 generations.

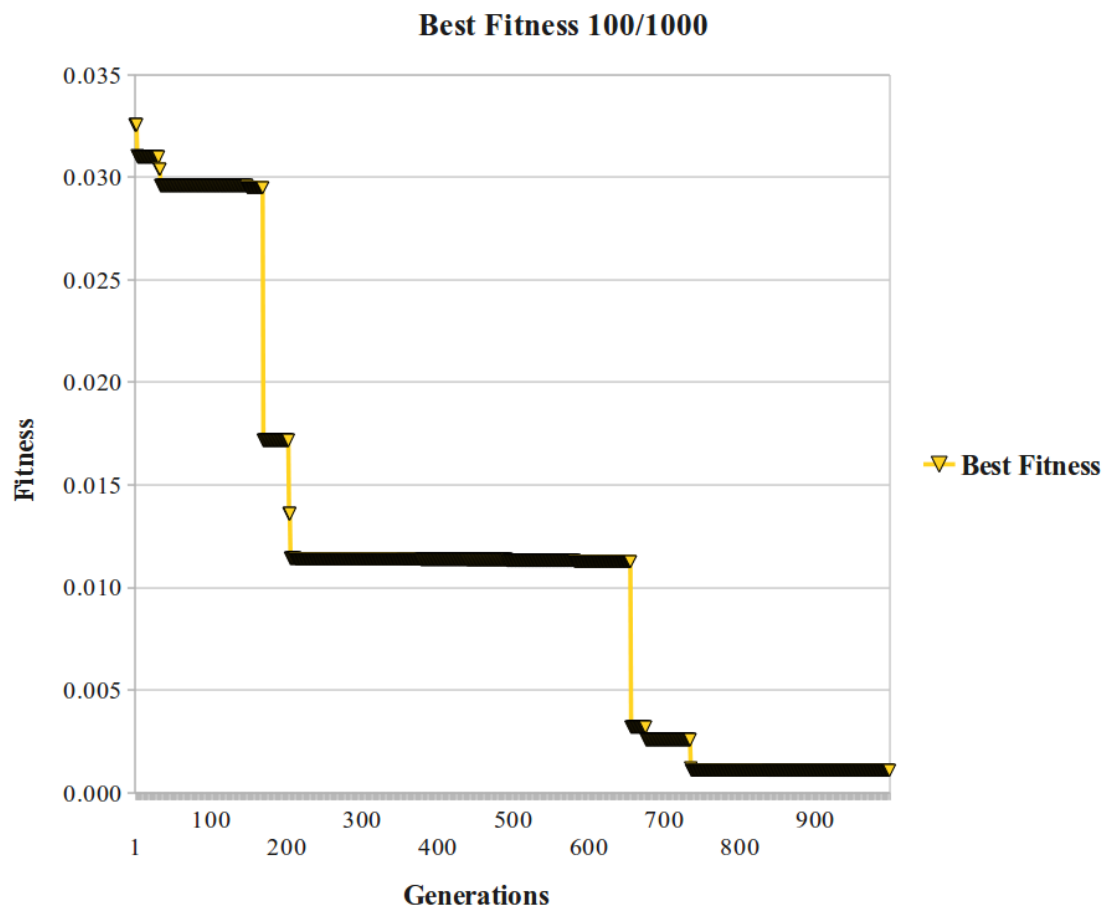


Figure 1. *Best fitness for 1000 generations*

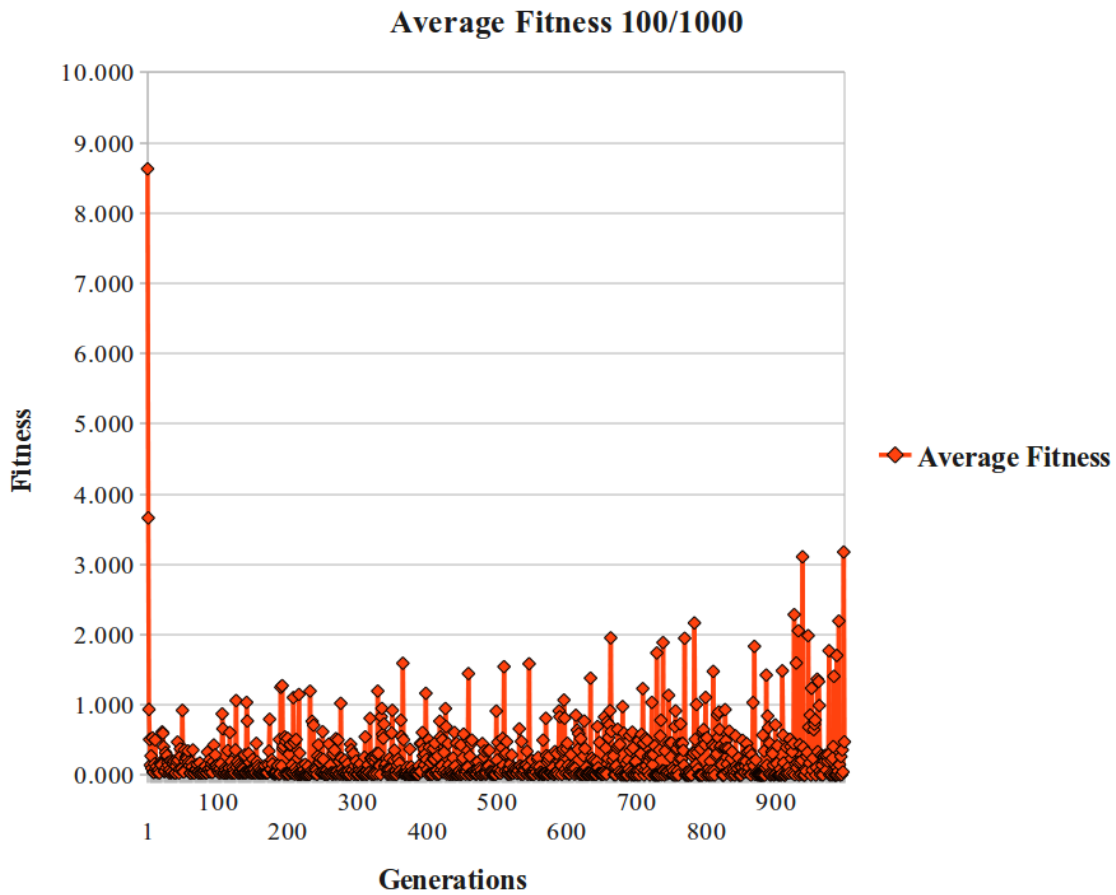


Figure 2. *Average fitness for 1000 generations*

While it is expected that the algorithm produces an optimal solution which can be seen in Fig. 1, it also should be observed that the average fitness will also begin to converge towards the global minimum as the population begins to saturate with the fittest solutions, as shown in Fig. 2. The algorithm parameter settings for this execution of the GA are shown in Table 1.

Table 1. *GA parameter settings*

<b>Population Size</b>	100
<b>Probability of Crossover</b>	60%
<b>Probability of Mutation</b>	5%
<b>Number of Generations</b>	1000

## CHAPTER III

### METHODOLOGY AND DEVELOPMENT OF A PARALLEL GENETIC ALGORITHM IN C++

#### **Parallelization methodology and the OpenMP API**

##### *A brief history of the OpenMP specification*

The OpenMP library specification was originally designed with the intent to bring serialized code to the parallel forefront *with portability*, or to overcome the ensuing paradigm obstacles that occurred as a result of the diversity among different parallel programming implementations. To unify a single model, the OpenMP Architecture Review Board (ARB) was set up circa 1997 to specify and maintain a standard implementation and protocol (Chapman, Jost, and Van Der Pas, 2008). Though OpenMP is a modest library compared to the extensive libraries such as those found in the Message Passing Interface (MPI) and the Threading Building Blocks (TBB), the feature set consists of enough preprocessor directives, library functions, and environment variables to exploit potential parallelism within serial code.

##### *What is OpenMP?*

Any software implementation, regardless of its type or application, consists of ‘code’ programmed in a language, or set of languages, as selected by the programmer. This code serves as the syntactical construct from which the software is built, and must call libraries of functions, data structures, and objects, to follow specified algorithms and

calculations. OpenMP is a library designed for implementation of compiler directives to add parallelization to the resultant machine code (that is, code that has been compiled and linked). OpenMP is an Application Programming Interface, or API. OpenMP works by creating and distributing threads, or runtime entities that are able to independently execute a stream of instructions (Chapman, Jost, and Van Der Pas, 2008). This distribution is done through a ‘fork and join’ model which is illustrated in Figure 3.

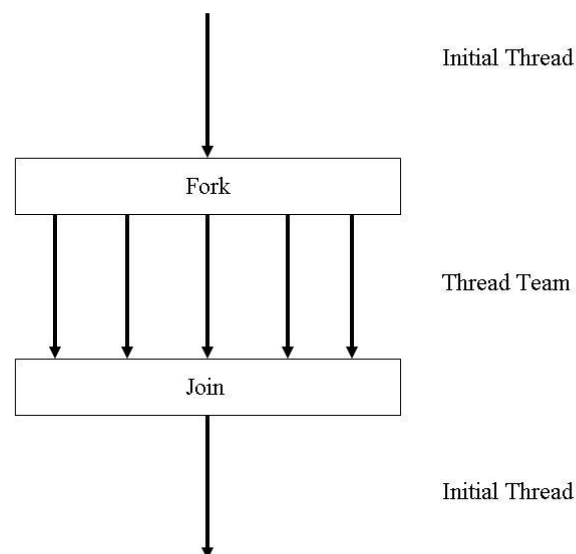


Figure 3. *Fork and join model*

In this model, threads are split by an OpenMP compiler directive off of the main initial thread into a team of threads that may be sent to other cores within a processor, or

different processors entirely. In code, the directive marks the beginning of a parallel region in which the thread team is created. When the code exits the parallel region, it rejoins the initial thread and the code continues to execute sequentially. This has two important implications. The serial code is left entirely intact (Chapman, Jost, and Van Der Pas, 2008), and unlike other parallel programming models, OpenMP does not require the entire code to be redesigned. This design also allows the code to be parallelized incrementally. The level of parallelization is left entirely up to the programmer, as code can be parallelized piece by piece, which also facilitates the debugging process.

#### *Identification of parallelizable regions of code*

The first step of parallelizing serial code is to evaluate the implementation to determine segments of code that could be converted into parallel processes. In order to be eligible for parallelization a code segment must have one of the following characteristics:

- Iteration independent loops
- Recurrence calculations
- Parallelized segments requiring thread control

To implement OpenMP, one does not require an understanding of all the underlying concepts of parallelization, but should understand the dangers of allowing threads to multiply across several cores within a machine. A thread is “a runtime entity that is able



to independently execute a stream of instructions” (Chapman, Jost, and Van Der Pas, 2008). For example, consider the following loop in Fragment 1.

```
for(i=0 ; i < 100; i++)
    array[i] = 45;
```

Fragment 1. *Example illustrating parallelizable for loop*

In this example, distributing multiple iterations across multiple threads (and consequently, possibly processing cores), is entirely harmless as all elements in 'array' are assigned the same value. Now consider Fragment 2.

```
for(i=0;i<100;i++)
    array[i] = array[i+1] + 45;
```

Fragment 2. *Example illustrating destroyed parallelism*

In the second example, the parallelism is destroyed simply because each successive iteration wholly depends on the next element within the array. Should this fragment be parallelized, it would cause problems for two reasons. Firstly, since threads run independently, the value of array [i+1] might have one thread attempting to access it and another thread attempting to update it simultaneously. Secondly, multiple threads may simply attempt to access the same value simultaneously. These kinds of bugs tend to fail 'silently' (i.e. without crashing the program, but causing data to become corrupt), and can therefore be difficult to debug.

To overcome problems such as this one, care must be taken in both the selection of functions and methods to be parallelized (e.g. or what algorithms should be used or avoided entirely), and how to implement the correct feature at the correct location.

### **The feature set**

A brief summary of the features that are implemented in the parallelization of the GA is described here. The parallelization utilizes the full OpenMP specification, as detailed at [www.openmp.org](http://www.openmp.org).

The following constructs are utilized:

*Parallel construct* - `#pragma omp parallel clause(---)`

This construct prompts the compiler that the code that follows should be executed in parallel. This directive serves as the foundation for additional optional OpenMP extensions and therefore is an indispensable part of the API (Chapman, et al., 2008).

*Parallel construct clauses*

- `if(scalar expression)`

This clause is particularly useful in particular implementations where it may be advantageous to parallelize a region based on the overhead expense, which can be evaluated based on a scalar value. This construct

provides in situ checking to allow the program to pseudo-optimize itself.

In practice, this works as a standard 'if' statement.

- `num_threads(integer expression)`

This clause is utilized to specify an explicit number of threads that should be used to execute any given parallel region. It is capable of overriding the `omp_set_num_threads` runtime function.

- `private(list)`

This clause enables (and forces) parallelized loops to keep track of their own iterations and prevent the aforementioned race condition.

- `firstprivate(list)`

This clause sets the variables in the list as private variables, but assigns them the same value as the variables with identical names before the construct.

- `lastprivate(list)`

A primary drawback of the private clause is that the value of the variable is destroyed immediately following the corresponding parallel region.

The `lastprivate` clause is an implemented workaround for this type of

scenario, effectively, `lastprivate` saves this attribute and allows for access after the construct.

- `shared(list)`

This clause explicitly defines which variables are to be shared across each thread. This is a prime instigator of race conditions and extreme care should be exercised when using this clause. OpenMP does not provide any type of race condition checking, and places that responsibility on the programmer.

- `default(none|shared)`

This clause specifies the variable type, either `none` or `shared` can be included in the list. When `default(none)` is placed within a corresponding OpenMP statement, the compiler is essentially told that every variable in the following region will be explicitly specified (i.e. as *private*, *lastprivate*, etc). When `default(shared)` is implemented, the compiler treats every variable that is not explicitly stated otherwise, as `shared`. It is the author's recommendation, from programming the algorithm, to avoid this, as it is a prone cause of segmentation faults.

*For construct* - `#pragma omp for clause(---)`

This construct is responsible for the parallelization of the for-loop. Given that most

programs spend a majority of computational time executing code within loops, this construct can prove to be particularly useful. In this section is also mentioned the ability of the OpenMP standard to allow for combined constructs. Examples are provided.

- `nowait`

This clause has no arguments, but removes the implied barrier that exist at the end of every parallel execution region. This allows threads to execute another instruction stream upon completing their current instruction stream.

### **Environment variables**

OpenMP requires a couple environment variables to be set either before or at runtime. Additionally, the specification also provides a set of variables that do not need to be set at runtime, but that offer additional support for the fine tuning of parallel programs. The two environment variables used in this study are:

- `OMP_DYNAMIC`

This environment variable allows for the compiler to adjust the number of threads at runtime. This variable operates as a flag, and specifies if dynamic thread allocation is allowed (e.g. `false` implies that every parallel region must utilize N threads where N is the number set by

OMP\_NUM\_THREADS variable, true implies that the number of threads can be changed).

- OMP\_NUM\_THREADS

This variable specifies the integer used to initialize the control variable *nthreads-var* (note, control variables are not discussed here).

### Environment subroutines

The following is an extracted set of subroutines offered by the OpenMP specification used to modify the environment or control variables.

- (int) omp\_get\_max\_threads(void)

This function returns the maximum allowable threads that can be used in the next encountered parallel region.

- (void) omp\_set\_num\_threads(integer expression)

This function can be used to override the OMP\_NUM\_THREADS environment variable by manipulating the underlying control variable *nthreads-var*, directly. This can provide a particularly useful avenue for optimization, as the author noted, when used in conjunction with `omp_get_max_threads()`, as the maximum available can be used to parallelize particularly computational intensive parallel regions.

- omp\_set\_dynamic(expression)

The function, like the aforementioned environment variable, is

responsible for manipulating the API to allow for dynamic thread allocation. If *expression* evaluates to *true* then the any number of threads up to the number specified by *nthreads-var* can be used. In practice, utilizing this subroutine may introduce a severe performance penalty.

- *(logical expression)* `omp_get_dynamic(void)`

This function is primarily useful for the first initial test runs with parallelized code. When called, `omp_get_dynamic` will check to determine the current state. If the state allows for dynamic thread allocation then it returns *true* ,it returns *false* otherwise.

- *(logical expression)* `omp_in_parallel(void)`

This function is particularly useful, in debugging parallelized code. This subroutine returns *true* if the region from which it is called is a parallel region, it returns *false* otherwise.

## Parallelization approach for a genetic algorithm

The objective of this research is to partially parallelize the genetic algorithm. If the algorithm were to be parallelized entirely, then much time would be wasted on the creation of new threads rather than on completing the task itself. This research has selected regions of code that should be parallelized based on the approximate execution time of the region *in comparison* to the amount of time it takes to create a new thread, copy private variables, execute code, and rejoin to the master (i.e. how long it takes to ‘fork’ and ‘join’). Only regions that take up the *most* amount of time were parallelized to illustrate that even exploiting the most obvious regions of code can have a tremendous impact on performance. To solidify this into an approach, *regions with strong repetition and long iterative loops should be the first targets in parallelization of an algorithm*. It follows that in algorithm development, subroutines demonstrating highly exploitable parallelism should be selected whenever possible, especially if they are responsible for executing repetitive regions of code.

Another important aspect to consider when approaching the design of a large-scale algorithm is to keep in mind the network (hardware) on which the algorithm will be operating on. A foundational understanding of shared memory systems (e.g. cluster versus a desktop computer) will help tremendously in the creation of parallel code. As an example, this algorithm was designed for a desktop machine. For a cluster implementation, it may be wiser to send the entirety of the code to be replicated on each node. A cluster version of an algorithm will most likely contain larger regions of



parallelized code due to more computational power available for that particular parallel region. The hardware on which the algorithm is to be run should play a role in the algorithms development at the pseudo-code level. This means subroutine selection must be imminent for parallelization. The source code for the GA was examined to identify areas that have exploitable parallelism. The global variable `POPULATION_SIZE` is the largest variable that appears in loop constructs, and these loops are targeted for parallelization. Any loop that is executed over `POPULATION_SIZE` executes the same operation for every individual in the population and exhibits inherent parallelism that is exploited in this implementation.

## CHAPTER IV

### SERIAL RESULTS

The serial code was executed for the test problem for a set of runs to determine and evaluate the overall accuracy of the algorithm. A set of trials were used to test the algorithm and to select appropriate parameter settings. Algorithmic settings that were determined *a priori* are:

Population Size = 100

Probability of Crossover ( $P_c$ ) = 60%

Probability of Mutation ( $P_m$ ) = 5%

Number of Elites = 2

Three settings for the number of generations were tested: 250, 500, and 750 generations. The convergence of the average fitness across the population for each of these setting is shown in Figs. 4, 5, and 6, respectively. Ideal convergence should show a more sloping graph towards the final generations, but the mutation rate of 5% maintains a significant amount of diversity in the population. The convergence of the best individual for each setting is shown in Figs. 7, 8, and 9, for 250, 500, and 750 generations, respectively. The global minimum for the test function is at (1,1) and has a fitness of zero. For each of the three generation settings, the GA is able to identify a nearly-optimal solution.

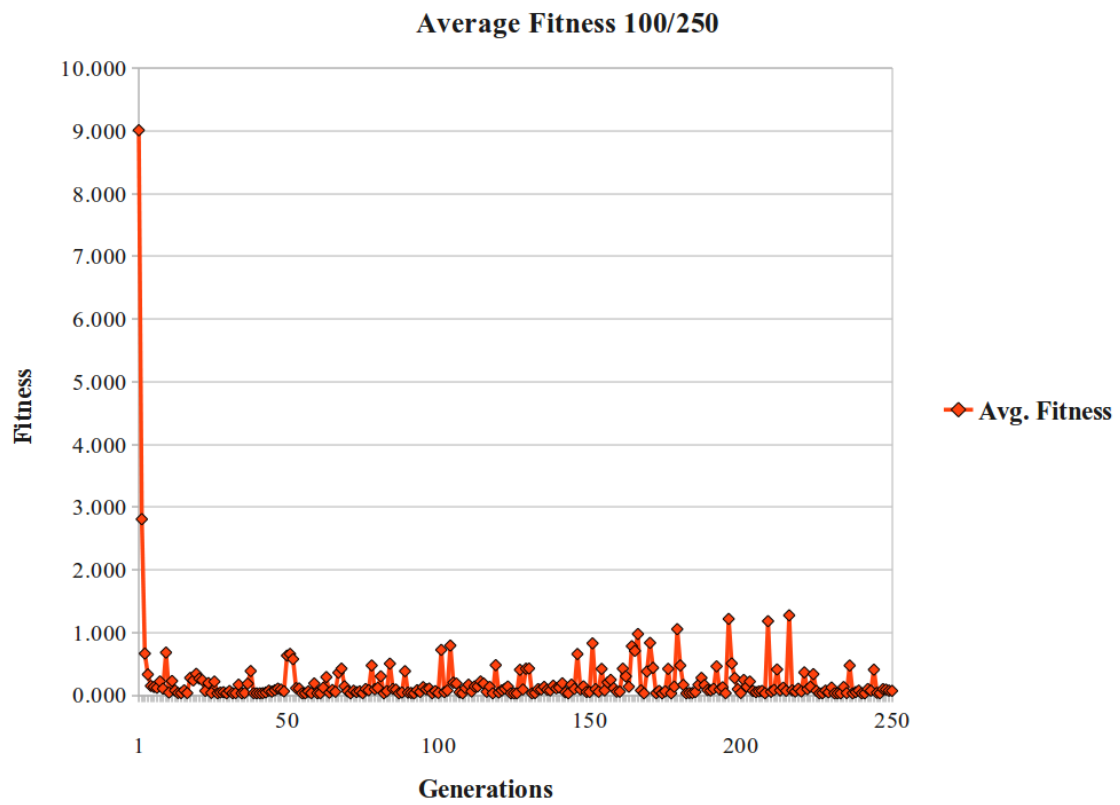


Figure 4. *Average fitness for 250 generations*

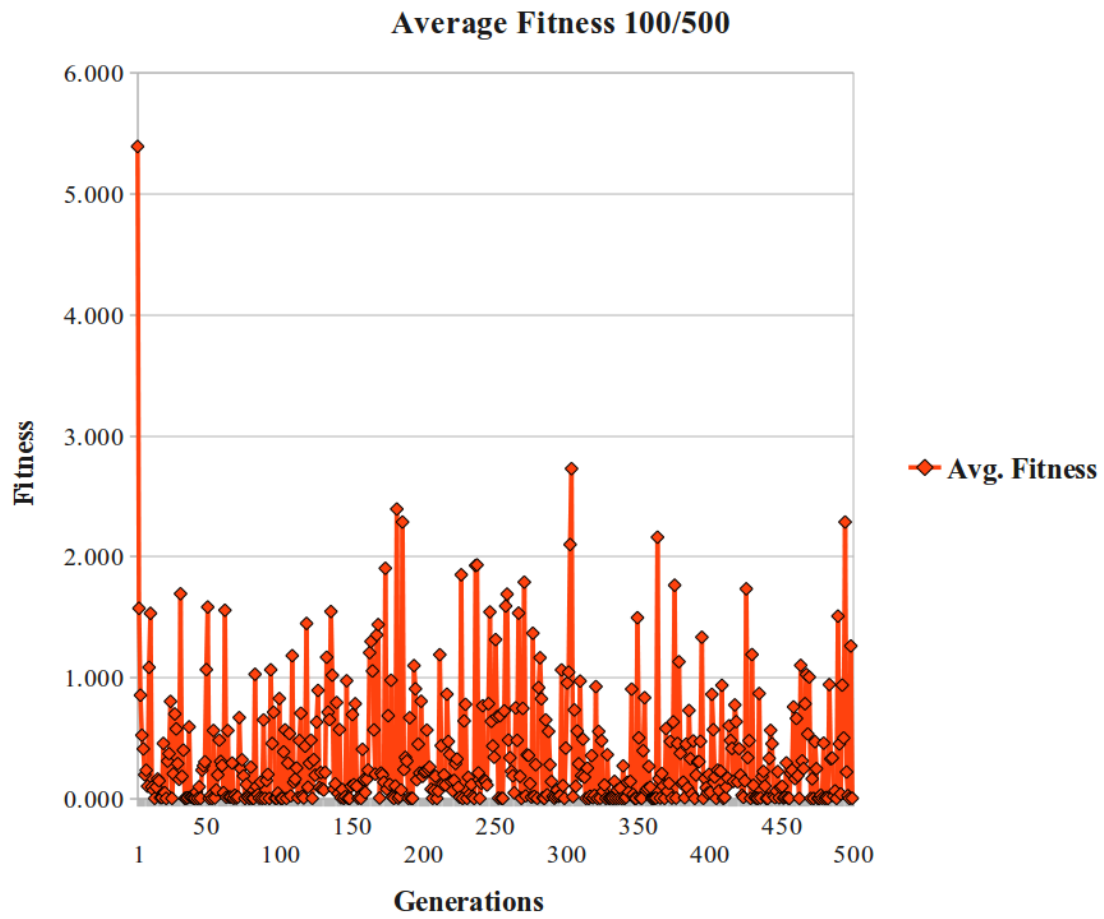


Figure 5. *Average fitness for 500 generations*

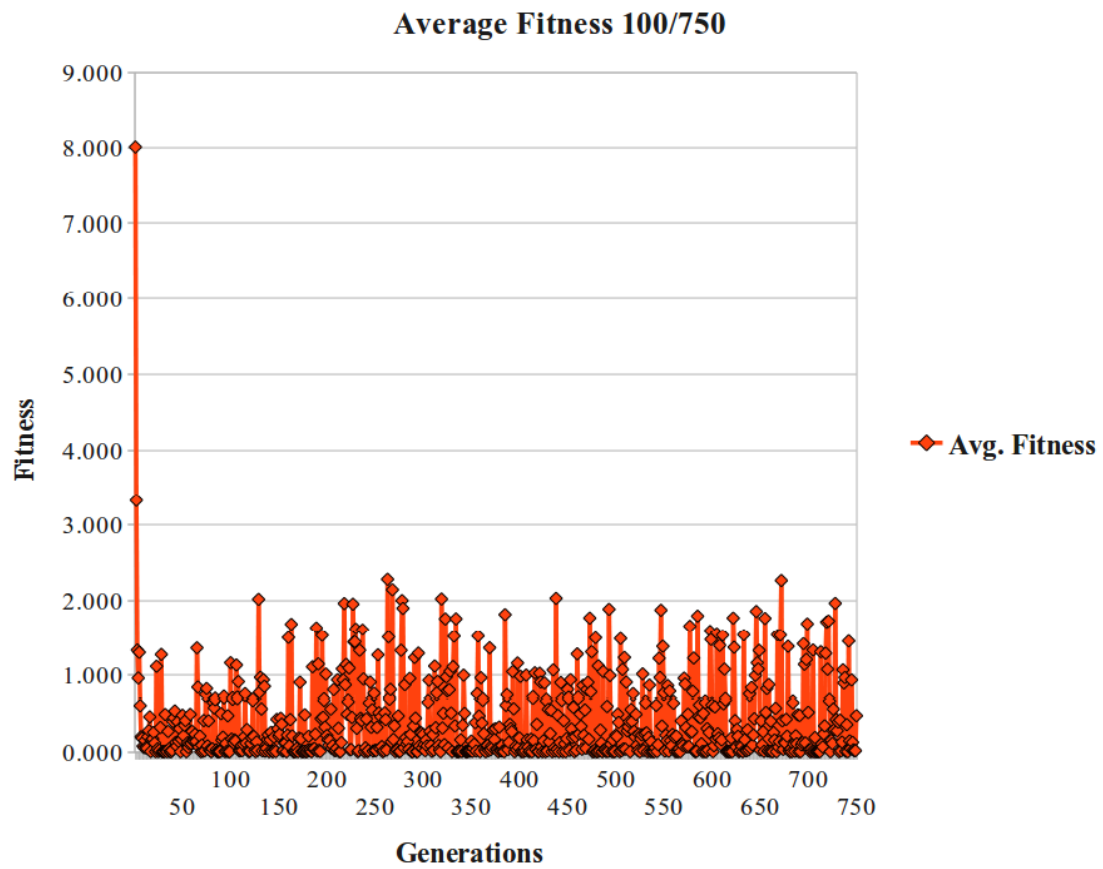


Figure 6. *Average fitness for 750 generations*

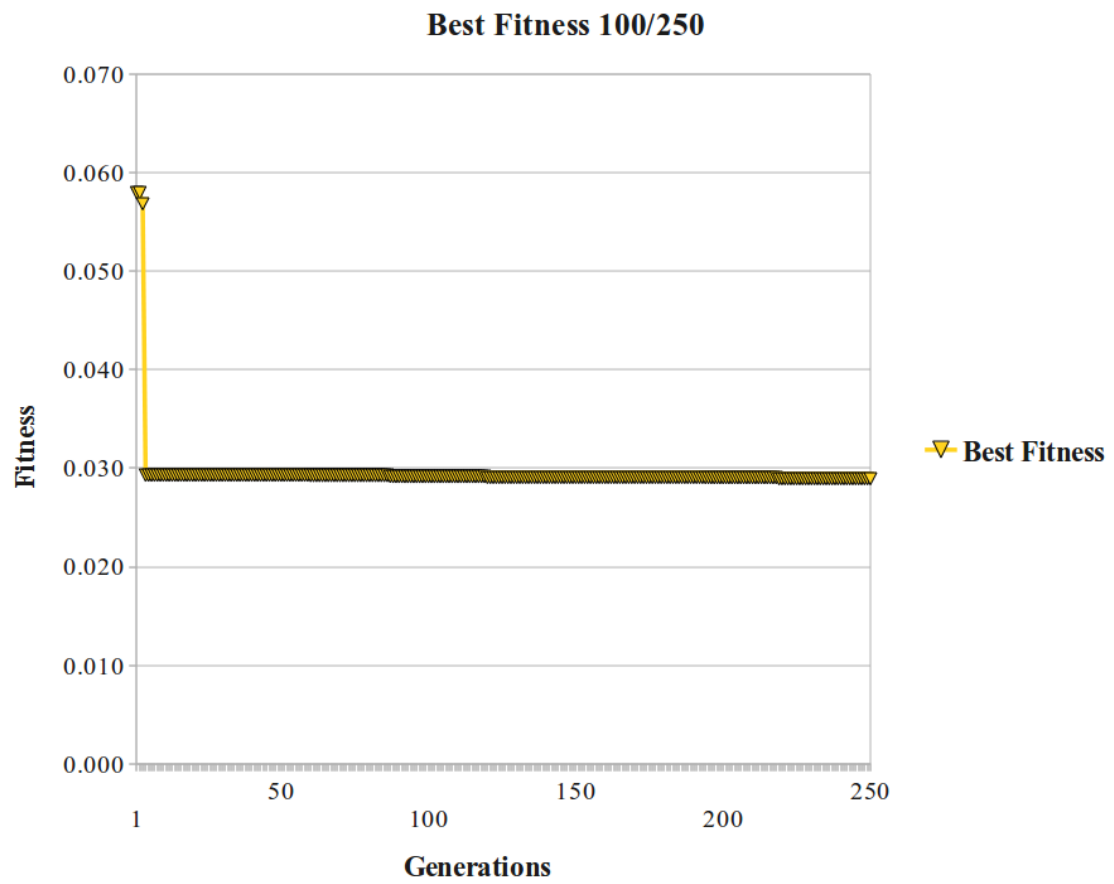


Figure 7. *Best fitness for 250 generations*

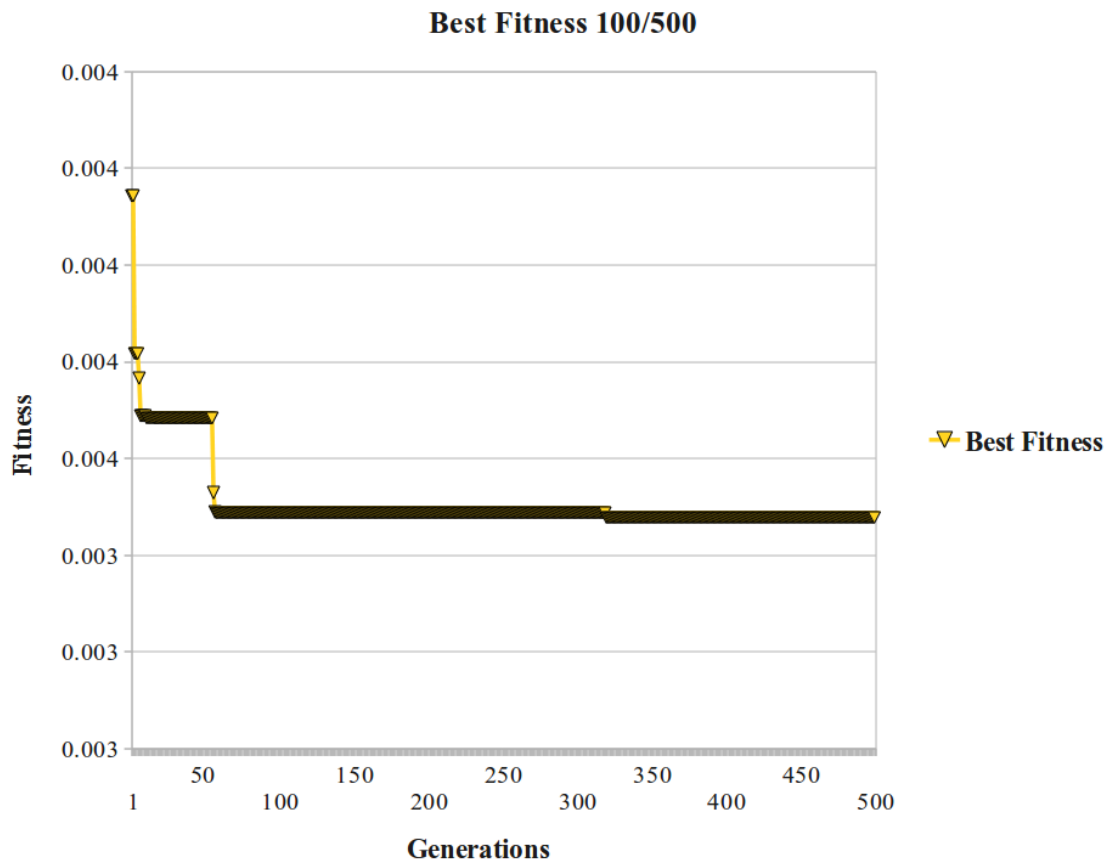


Figure 8. *Best fitness for 500 generations*

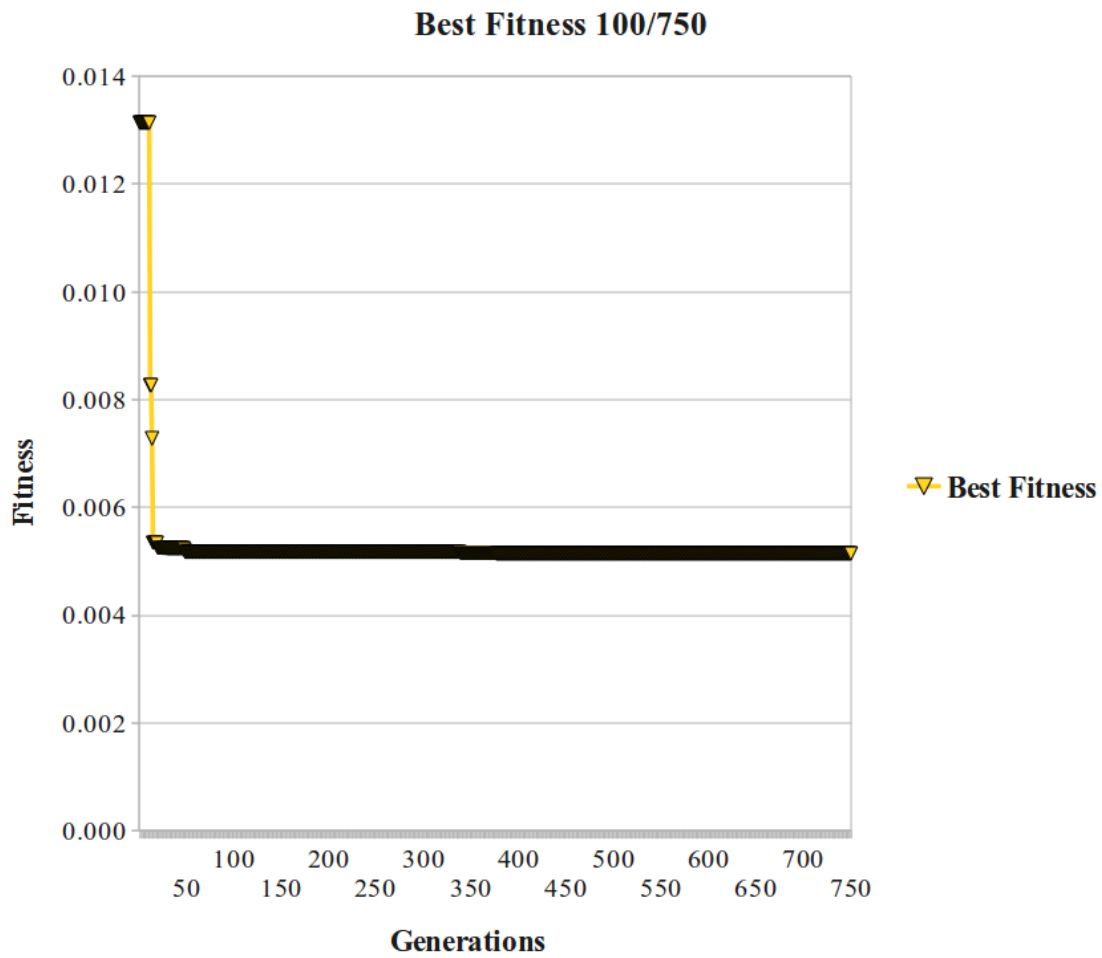


Figure 9. *Best fitness for 750 generations*

As a stochastic heuristic algorithm, a GA should be executed for a set of trials to test its ability to consistently identify the global optima. The best solution out of 20-30 runs can be identified as the best solution. The GA was run with various parameter settings for 30 runs (Table 2).



Table 2. *Serial performance with different parameter settings*

Serial GA Statistics									
<u>Parameters</u>	Normal	Suite 1	Suite 2	Suite 3	Suite 4	Suite 5	Suite 6	Suite 7	Suite 8
Runs	30	30	30	30	30	30	30	30	30
Generations	500	500	500	500	500	500	500	500	500
Population Size	100	100	100	100	100	75	75	75	75
Pc	60	70	60	70	60	60	70	70	75
Pm	5	5	7	7	10	10	5	10	5
Elites	2	3	1	3	2	2	3	2	3
<u>Analysis</u>									
Average Fitness AAR	0.3610	0.9122	1.2684	0.4621	1.5044	1.0789	0.1887	1.1350	0.2037
Average Best Fitness AAR	0.0052	0.0015	0.0030	0.0034	0.0008	0.0031	0.0056	0.0013	0.0000
Standard Deviation AFAAR	0.05217	0.04915	0.04734	0.04136	0.05621	0.04237	0.05671	0.05397	0.05310
Standard Deviation BFAAR	0.00038	0.00087	0.00033	0.00007	0.00015	0.00278	0.00024	0.00027	0.00003
Best Fitness	8.736E-05	4.230E-06	1.412E-03	2.581E-03	3.181E-05	5.759E-04	1.753E-04	3.083E-04	9.072E-05
Best Allele 0	0.991	0.998	0.962	0.949	0.994	0.976	0.987	0.982	0.990
Best Allele 1	0.981	0.996	0.926	0.901	0.989	0.953	0.974	0.965	0.981
Runtime (seconds)	2738.65	2739.72	2736.81	2736.07	2736.09	2065.68	2063.50	2070.13	2085.63
Average Fitness Across All Runs (AFAAR)									
Best Fitness Across All Runs (BFAAR)									

Suites 1,4, and 8 acquired the best solutions (although every Suite was able to find a values fairly close to the global minimum), all of which had a Pc value of at least 70%.

The difference between Suites 3 and 4 indicates that re-injecting the elite individual three times may have some advantages over only re-injecting the elite individual twice. Suite 6 had the lowest average fitness across all runs (AAR) indicating that it exhibited the most convergence towards the global minimum, all solutions considered. This is interesting as it also had 25% fewer solutions contained within its population. Suite 3 had the lowest standard deviation in average fitness AAR indicating that a majority of its solutions tended to show similar characteristics to its elite individual, and Suite 8

performs better than most suites, showing dominance in average fitness AAR, average best fitness AAR, and standard deviation of the best fitness AAR. Suite 8 used a population of 75, the highest percentage for crossover among the Suites, (75%), a moderate level of mutation (5%), and three elite solutions saved for each of 500 generations.

## CHAPTER V

### PARALLEL RESULTS

Loops that contained the global variable POPULATION\_SIZE were chosen for parallel implementation. Code fragments illustrating this for Initialization, Evaluation, Selection, Crossover, and Generation Switch are shown in Fragments 3-7.

```
//Initialization Operator
void Population::Populate()
{
    MTRand random_gen;
    int i;
    double r1,r2;
    omp_set_num_threads(2);
    #pragma omp parallel for default(shared) private(i,r1,r2)
    for(i=0;i<POPULATION_SIZE;i++)
    {
        r1 = random_gen.rand();
        r2 = random_gen.rand();
        this->group_of_individuals[i].genes[0] = r1;
        this->group_of_individuals[i].genes[1] = r2;

        this->group_of_individuals[i].fitness =
group_of_individuals[i].evaluate(group_of_individuals[i].genes[0],group_of_individuals
[i].genes[1]);
    }
};
```

Fragment 3. *Parallelized initialization operator*

As can be seen in Fragment 3, inserting the preprocessor directive is fairly straightforward (immediately above the for loop); however, care must be taken in

ensuring the variable types (e.g. *private*, *shared*, *etc*) are correct. In this context, the variable of iteration (iterator), *i*, is declared private so that each thread has a local copy – this prevents threads from simultaneously accessing the same memory location for *i*. In the following example (Framgent 4), the *nowait* clause is introduced.

```
//Evaluate
void Population::Evaluate(Individual *elite)
{
    int i,j;
    j = POPULATION_SIZE;
    #pragma omp parallel shared(j) private(i)
    #pragma omp for nowait
    for(i=0;i<j;i++)
    {
        group_of_individuals[i].fitness =g roup_of_individuals...;
        next_group_of_individuals[i].fitness = next_group_of_individ...;
    }
}
```

Fragment 4. *Parallelized evaluate operator*

The parallelization can be carried out much the same way for other operators, as shown in Fragments 5-7.

```

//Tournament Selection
//Tournament Vars
Individual Pa, Pb;
double rdm_decide;
int rdm1, rdm2,j,s;
s = POPULATION_SIZE;
#pragma omp parallel shared(s) private(j)
#pragma omp for nowait
for(j=0;j<s;j++)
{
    rdm1 = (int)floor( (random_gen.rand()*POPULATION_SIZE) + 0.5);
    rdm2 = (int)floor( (random_gen.rand()*POPULATION_SIZE) + 0.5);
    rdm_decide = random_gen.rand();
}

```

Fragment 5. *Parallelized selection operator*

```

void Population::Crossover()
{
    MTRand random_gen;
    Individual Pa,Pb,Child1,Child2;
    int random,random2,random3,i,x;
    x = POPULATION_SIZE;
    #pragma omp parallel shared(x) private(i)
    #pragma omp for nowait
    for(i=0;i<x;i++)
    {
        // random definitions
        random = (int)floor( (random_gen.rand()*POPULATION_SIZE) + 0.5);
        random2 = (int)floor( (random_ge...;

```

Fragment 6. *Parallelized crossover operator*

```

void Population::GenerationSwitch()
{
    int i,x;
    x = POPULATION_SIZE;
    //max = omp_get_max_threads();
    //omp_set_num_threads(max);
    #pragma omp parallel shared(x) private(i)
    #pragma omp for nowait
    for(i=0;i<x;i++)
    {
        // This will also be affected by my question in "Selection"
        // Copy 'next' into the primary population being 'worked on'
        group_of_individuals[i].genes[0] = next_group_of_individuals[

```

Fragment 7. *Parallelized generation switch operator*

For these Fragments, the entire operator is not shown, and comments have been deleted.

The code in its entirety is housed in Appendix C.

## Optimization

Once the algorithm was able to run with a small degree of parallelism, fine tuning the parameters was necessary to minimize the overhead. Parameters, such as the number of variables to declare as shared and private, impact both performance and accuracy of the algorithm. Parameters such as the specification of a critical region and decisions to parallelize a region played a major role in determining the algorithm's overall runtime. For example, the mutation operator was not selected for parallelization, due to the incurred overhead that would have resulted from parallelizing an operator that is executed for only 5-10% of the total population. Each successive OpenMP directive comes at a performance penalty. Before a region is parallelized, the performance gain should be verified. Through trial and error, regions were selected for parallelization

based on their ability to achieve speedup.

To illustrate, consider the following runtimes for the different parameter Suites in Table 2. The code that yielded the fastest runtime (Suite 1) is included in Appendix A, and all runs were conducted on the following machine:

**Motherboard:** MSI – P35 Neo 2

**Processor:** Intel Core 2 Duo E6750 @ 3.4 GHz

**RAM:** ADATA 2 GB @ 850 MHz 5-5-5-18

**HDD:** WD 200 GB 7200 RPM SATA II

**OS:** Ubuntu 9.10

Results of parallelization are shown in Table 3. A performance metric is given by Equation 2 (Patterson and Hennessy, 2009).

$$\frac{\text{Algorithm Performance A}}{\text{Algorithm Performance B}} = \frac{\text{Execution Time B}}{\text{Execution Time A}}$$

Equation 2. *Equation for calculating speedup*

Table 3. *Runtimes and OpenMP parameters for parallel implementation*

Parallel GA Statistics								
<u>Parameter</u>	<u>Serial</u>	<u>Suite 1</u>	<u>Suite 2</u>	<u>Suite 3</u>	<u>Suite 4</u>	<u>Suite 5</u>	<u>Suite 6</u>	<u>Suite 7</u>
Dynamic Thread Allocation	N/A	No	No	Yes	No	Yes	No	No
Number of Threads	1	2	3	3	Adjusted	Adjusted	Adjusted	2
nowait clause used	N/A	No	No	No	No	Yes	Yes	Yes
Initialization Optimized	N/A	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Evaluate Optimized	N/A	Yes	Yes	Yes	Yes – 2 Threads	Yes – 2 Threads	Yes – 2 Threads	Yes
Selection Optimized	N/A	Yes	Yes	Yes	Yes – 3 Threads	Yes – 3 Threads	Yes – 3 Threads	Yes
Crossover Optimized	N/A	Yes	Yes	Yes	Yes – 3 Threads	Yes – 3 Threads	Yes – 3 Threads	Yes
Generation Switch	N/A	Yes	Yes	Yes	Yes – Max. Threads	Yes – Max. Threads	Yes – Max. Threads	Yes
Runtime (seconds)	181	112	132	180	144	181	143	115
Speedup	1	1.62	1.37	1.01	1.26	1	1.27	1.57

From Table 3 it can be shown that OpenMP can clearly offer a performance increase demonstrating a significant speedup over the serial implementation. Using Equation 2 and comparing the serial implementation with Suite 1, a speedup of 1.6 is observed.

Algorithmic parameters much be tuned, for example which variables should be declared as *shared* or *private*. Poor parallelization can lead to insignificant performance gains, or even to performance loss, as can be seen by Suites 3 and 5.



## **CHAPTER VI**

### **CONCLUSIONS**

From this study, the importance of utilizing parallelizable subroutines has been identified. Conceptualizing which subroutines have exploitable parallelism during the pseudo-code stages of development will prove to be increasingly valuable for problems of increased size and complexity. Accuracy, however, should never be lost due to parallelism. In some cases, an algorithm may simply be in need of restructuring to exploit whatever parallelism it may have to offer. Building algorithms based on accuracy foremost, and parallelism secondly, when coupled with the additional design considerations of robustness and user-specific needs, provides a more expandable foundation to better anticipate the problems of the future.

Algorithms that are characterized by loops with strong iteration dependency (e.g. Fragment 2) should be avoided or restructured. A multiplicity of independent tasks can be safely broadcasted over a large number of threads, which renders evolutionary algorithms excellent candidates for parallelization. Other algorithms, for example, linear programming, typically contain elements that simply cannot be decoupled from their iteration dependent nature.

In the future, cross-compilers may be available to restructure algorithms with poor parallelism into those that exhibit better parallelism, but this task should not be expected

from an API. A specification such as OpenMP should not be expected to resolve poor subroutine selection, and this task falls to the programmer, due to the creativity and analysis needed to identify appropriate subroutines. Attempting to parallelize source code far into the developmental stages may result in sub-optimal parallelization schemes, and the design for parallelization should be well-thought out at early stages. Given the increasing availability of multi-core hardware and computing clusters, the development of large scale algorithms for parallel environments may be exploited to increasingly higher scales and for more applications in the future.

From this study, OpenMP proves itself to offer significant advantages for improving the performance of a genetic algorithm. The computational gains indicate a 1.6 speedup, and the performance of the algorithm to identify fit solutions is maintained. In the future, further research will apply this implementation for large engineering design problems to take advantage of the savings in computational time and identify efficient design strategies.

## REFERENCES

- Battiti, R., and Tecchiolli, G. (1992). "Parallel Biased Search for Combinatorial Optimization: Genetic Algorithms and TABU." *Microprocessors and Microsystems*, 16(7), 351 - 367.
- Cantu-Paz, E., and Goldberg, D.E. (1999). "On the Scalability of Parallel Genetic Algorithms." *Evolutionary Computation*, 7(4), 429 - 449.
- Chapman, B., Jost, G., and Van Der Pas, R. (2008). "Using OpenMP, Portable Shared Memory Parallel Programming." (*Scientific and Engineering Computation Series*). The MIT Press. Boston, MA.
- Ewald, G., Kurek, W., and Brdys, M.A. (2008). "Grid Implementation of a Parallel Multiobjective Genetic Algorithm for Optimized Allocation of Chlorination Stations in Drinking Water Distribution Systems: Chojnice Case Study." *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(4), 497 - 509.
- Holland, J. (1992). "Genetic Algorithms." *Scientific American*, 267(1), 66-72.
- Kandil, A. and, El-Rayes, K. (2006). "MACROS: Multiobjective Automated Construction Resource Optimization System." *Journal of Management in Engineering*, 22(3), 126-134.
- Kegel, P., Schellmann, M., and Gorlatch, S. (2009). "Using OpenMP Vs. Threading Building Blocks for Medical Imaging on Multi-Cores." *LNCS*, 5704, 654 - 665.
- Kuczera, G. (1992). "Water Supply Headworks Simulation Using Network Linear

- Programming." *Advances in Engineering Software*, 14(1), 55 - 60.
- Mirghani, B.Y., Mahinthakumar, K.G., Tryby, M.E., Ranjithan, R.S., and Zechman, E.M. (2009). "A Parallel Evolutionary Strategy Based Simulation–Optimization Approach for Solving Groundwater Source Identification Problems." *Advances in Water Resources*, 32(9), 1373-1385.
- Mitchell, M. (1996). *An Introduction to Genetic Algorithms. A Bradford Book* The MIT Press, Cambridge, MA.
- Muhlenbein, H., Schomisch, M., and Born, J. (1991). "The Parallel Genetic Algorithm as Function Optimizer." *Parallel Computing*, 17(6-7) , 619 - 632.
- The OpenMP Architecture Review Board (2008). "OpenMP Application Programming Interface: Version 3.0." < <http://www.openmp.org/mp-documents/spec30.pdf> >
- Patterson, D., and Hennessy, J.H. (2009). "Computer Organization and Design, Fourth Edition: The Hardware/Software Interface." *The Morgan Kaufmann Series in Computer Architecture and Design*, Morgan Kaufmann, Burlington, MA.
- Rausch, T., Thomas, A., Camp, N. J., Cannon-Albright, L. A. and, Facelli, J. C. (2008). "A Parallel Genetic Algorithm to Discover Patterns in Genetic Markers That Indicate Predisposition to Multifactorial Disease." *Computers in Biology and Medicine* 38(7), 826-36.
- Tendler, J.M., Dodson, J.S., Fields, Jr., J.S., Le, H., and Sinharoy, B. (2002). "POWER4 System Microarchitecture." *IBM Journal of Research and Development*, 46(1), 5 -26.
- The OpenMP Architecture Review Board (2008). "OpenMP Application Programming

Interface: Version 3.0."< <http://www.openmp.org/mp-documents/spec30.pdf>>

Wang, N., Tsai, C. and, Cha, K. (2009). "Optimum Design of Externally Pressurized Air

Bearing Using Cluster OpenMP." *Tribology International*, 42, 1180-1186.

Wang, P., Katz, D. S., and Chao, Y. (1997). "Optimization of a Parallel Ocean

General Circulation Model." *Proc. Supercomputing, ACM/IEEE 1997*

*Conference*, ACM/IEEE, San Jose, CA, 25-36.

Wang, P. and, Wu, X. (2002). "OpenMP Programming for a Global Inverse Model."

*Scientific Programming*, 10(3), 253-261.

## **APPENDIX A**

### **SOURCE CODE**

**first\_GA\_main.cpp**

```

/*
 * first_GA_main.cpp
 *
 *
 */
#include "Population.h"
#include <iostream>
#include <time.h>
#include <omp.h>
using namespace std;

#define GENERATIONS 500
#define NUM_RUNS 1
int main()
{
    int i,m;
    double t_diff;
    time_t start,end;
    time (&start);

    Population myPop;
    Individual elite;

    cout << "Starting Run..." << endl;
    cout << "Populating Solution Space..." << endl << endl;
    myPop.Populate();

    /* I need to create a population...and then evolve it*/
    cout << "Beginning Evolution..." << endl << endl;

    for(m=0;m<NUM_RUNS;m+=1)
    {
        for(i=0;i<GENERATIONS;i+=1)
        {
            myPop.Evaluate(&elite);
            myPop.ElitistSave(&elite);
            myPop.Selection();
            myPop.Crossover();
            myPop.Mutation();
            myPop.Evaluate(&elite);
            myPop.ElitistCompare(&elite);
            myPop.EchoStatistics(i,elite);
            myPop.GenerationSwitch();
        }
        if(NUM_RUNS>1)
        {
            myPop.Populate();
        }
    }
}

```



```
cout << endl << endl << "Run Completed!" << endl;
cout << "Best Alleles: " << elite.genes[0] << " " << elite.genes[1] << endl;
cout << "Best Fitness: " << elite.fitness << endl;
//Time Mechanism
time(&end);
t_diff = difftime(end,start);
cout << "Elapsed Time: " << t_diff << " seconds" << endl;
return 0;
}
```

**Population.cpp**

```

/*
 * Population.cpp
 *
 * Created on: Nov 6, 2009, started at 630AM...on a Saturday
 * IM SO EXCITED!!! Praise the Lord!! Dr. Zechman is awesome!!!
 * Thank you so much for this opportunity!!
 * Author: Avery A. White, CEEL 09'
 * and...Many thanks to Daniel Eng and Darel Booth
 * for all their guidance on the coding end89
 */
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <fstream>
#include <omp.h>
using namespace std;
#include "Population.h"

/*
 * MersenneTwister random number generator from
 * http://www-personal.umich.edu/~wagnerr/MersenneTwister.html
 * see .h file for complete reference list
 */
#include "MersenneTwister.h"
// #include "Individual.h"

// I want to give my program the ability to create and destroy the class
// initializes all variables

Population::Population()
{
    int i;
    for(i=0;i<POPULATION_SIZE;i++)
    {
        group_of_individuals[i].genes[0] = 0;
        group_of_individuals[i].genes[1] = 0;
        group_of_individuals[i].fitness = 0;
        next_group_of_individuals[i].genes[0] = 0;
        next_group_of_individuals[i].genes[1] = 0;
        next_group_of_individuals[i].fitness = 0;

    }
    bst_indi = 0;
    avg_fitness = 0;
}

Population::~Population(){}

//Initialization Operator
void Population::Populate()

```

```

{
    MTRand random_gen;
    int i;
    double r1,r2;
    omp_set_num_threads(2);
    #pragma omp parallel for default(shared) private(i,r1,r2)
    for(i=0;i<POPULATION_SIZE;i++)
    {
        r1 = random_gen.rand();
        r2 = random_gen.rand();
        this->group_of_individuals[i].genes[0] = r1;
        this->group_of_individuals[i].genes[1] = r2;

        this->group_of_individuals[i].fitness =
group_of_individuals[i].evaluate(group_of_individuals[i].genes[0],group_of_individuals[i].genes[1]);
    }
};

//Evaluate
void Population::Evaluate(Individual *elite)
{
    int i,j;
    j = POPULATION_SIZE;
    #pragma omp parallel shared(j) private(i)
    #pragma omp for nowait
    for(i=0;i<j;i++)
    {
        group_of_individuals[i].fitness =
group_of_individuals[i].evaluate(group_of_individuals[i].genes[0],group_of_individuals[i].genes[1]);
        next_group_of_individuals[i].fitness =
next_group_of_individuals[i].evaluate(next_group_of_individuals[i].genes[0],next_group_of_individuals[
i].genes[1]);
    }
    elite->fitness = elite->evaluate(elite->genes[0],elite->genes[1]);
};

/* The following code is from:
* http://www.phys.unsw.edu.au/~mcba/phys2020/notes/sort.html
*/
// myComparisonFunction by Michael Ashley
// http://www.phys.unsw.edu.au/~mcba/phys2020/notes/sort.html
int myDoubleComparisonFunction(const void *x, const void *y) {

    // x and y are pointers to doubles.

    // Returns -1 if x < y
    //      0 if x == y
    //      +1 if x > y

    double dx, dy;

    dx = *(double *)x;

```

```

dy = *(double *)y;

if (dx < dy) {
    return -1;
} else if (dx > dy) {
    return +1;
}
return 0;
}

/* Heuristic (cool word huh?) method of selection
 * For those NOT familiar with the crazy terminology of this stuff...
 * I'm trying to 'code' a little bit of nature...in a sense. Just as nature's
 * algorithm for 'natural selection' aka what animals get eaten,
 * is seemingly random at times. I replicate that behavior here
 * by letting the computer 'decide' which individuals to 'eat' (replacement)
 * and therefore, which ones survive. Fitness is thrown in as a means to weight
 * the probability. That is, the more fit, the more likely to survive.
 *
 * This function is responsible for selecting individuals from my
 * current population and inserting them into the next population.
 * It does this by creating a random number, and then selecting
 * two individuals at random. If the random number is less than 0.95
 * (which will be the case more often than not) it substitutes
 * the more fit individual out of the two and inserts it into the
 * next generation. If the number is greater than the less fit
 * individual is inserted into the population.
 */
void Population::Selection()
{
    MTRand random_gen;
    //Tournament Selection
    //Tournament Vars
    Individual Pa, Pb;
    double rdm_decide;
    int rdm1, rdm2, j, s;
    s = POPULATION_SIZE;
    #pragma omp parallel shared(s) private(j)
    #pragma omp for nowait
    for(j=0; j<s; j++)
    {
        rdm1 = (int)floor( (random_gen.rand()*POPULATION_SIZE) + 0.5);
        rdm2 = (int)floor( (random_gen.rand()*POPULATION_SIZE) + 0.5);
        rdm_decide = random_gen.rand();

        //cout << "rdm_decide: " << rdm_decide << endl;
        Pa.genes[0] = group_of_individuals[rdm1].genes[0];
        Pa.genes[1] = group_of_individuals[rdm1].genes[1];
        Pa.fitness = group_of_individuals[rdm1].fitness;
        Pb.genes[0] = group_of_individuals[rdm2].genes[0];
        Pb.genes[1] = group_of_individuals[rdm2].genes[1];
        Pb.fitness = group_of_individuals[rdm2].fitness;
    }
}

```

```

        if(Pb.fitness > Pa.fitness)
        {
            // These values of 0.95 might be good to adjust to introduce more diversity in
            // As an experiment adjust the values of intervals of
            if(rdm_decide < 0.95)
            {
                next_group_of_individuals[j].genes[0] = Pa.genes[0];
                next_group_of_individuals[j].genes[1] = Pa.genes[1];
                next_group_of_individuals[j].fitness = Pa.fitness;
            }
            else
            {
                next_group_of_individuals[j].genes[0] = Pb.genes[0];
                next_group_of_individuals[j].genes[1] = Pb.genes[1];
                next_group_of_individuals[j].fitness = Pb.fitness;
            }
        }
        else
        {
            if(rdm_decide < 0.95)
            {
                next_group_of_individuals[j].genes[0] = Pb.genes[0];
                next_group_of_individuals[j].genes[1] = Pb.genes[1];
                next_group_of_individuals[j].fitness = Pb.fitness;
            }
            else
            {
                next_group_of_individuals[j].genes[0] = Pa.genes[0];
                next_group_of_individuals[j].genes[1] = Pa.genes[1];
                next_group_of_individuals[j].fitness = Pa.fitness;
            }
        }
    }
};

/*
 * On a purely bitwise level, I need to exchange the
 * bits of the individuals...sometimes
 * So when "Crossover" runs it's going to take in a variable
 * individuals' of type "Individuals"
 * Thanks to Darel Booth for helping me out on this one...
 *
 * Crossover as in "run the individual -level crossover a bunch of times"
 */
void Population::Crossover()
{
    MTRand random_gen;
    Individual Pa,Pb,Child1,Child2;
    int random,random2,random3,i,x;
    x = POPULATION_SIZE;
    #pragma omp parallel shared(x) private(i)

```

```

#pragma omp for nowait
for(i=0;i<x;i++)
{
    // random definitions
    random = (int)floor( (random_gen.rand()*POPULATION_SIZE) + 0.5);
    random2 = (int)floor( (random_gen.rand()*POPULATION_SIZE) + 0.5);
    random3 = (int)floor( (random_gen.rand()*POPULATION_SIZE) + 0.5);

    // Selecting Parent A and Parent B ~
    Pa = next_group_of_individuals[random];
    Pb = next_group_of_individuals[random2];

    Child1 = Pa.IndividualCrossover(&Pb);
    Child2 = Pb.IndividualCrossover(&Pa);

    /* If random variable 3 is less than the cross probability
     * then insert the children in randomly, else re-insert the parents
     */
    if(random3<PROB_CROSS)
    {
        next_group_of_individuals[random] = Child1;
        next_group_of_individuals[random2] = Child2;
    }
    else
    {
        next_group_of_individuals[random] = Pa;
        next_group_of_individuals[random2] = Pb;
    }
}

};

/*
 * The idea is to simulate nature's inexplicable random mutation technique...
 * to sort of 'cut to the chase' and find the most optimal solution ~ faster.
 * Checked: The numbers are indeed random.
 */
void Population::Mutation()
{
    //New Mutation Operator: Single Point Mutation with Simulated Annealing
    MTRand random_gen;
    Individual test;
    int random, random2,i;

    //Outer Loop Controls Number of Mutations
    for(i=0;i<((PROB_MUT/100.0)*POPULATION_SIZE);i++)
    {
        /* This produces a random number 0-100
         * Note that random is used to select from 'group' and
         * reinsert into 'next' generations. Since the number
         * generator is working to avoid repeating numbers
         * this should help by preventing recently mutated

```

```

    * (and therefore more fit then it's predecessor)
    * individuals from being immediately replaced on the
    * next iteration
    */
    random = (int)floor( (random_gen.rand()*POPULATION_SIZE) + 0.5);

    //Random number either 0 or 1
    random2 = (int)round(random_gen.rand());

    //Copy a random individual into test
    test.fitness = group_of_individuals[random].fitness;
    test.genes[0] = group_of_individuals[random].genes[0];
    test.genes[1] = group_of_individuals[random].genes[1];

    //randomly select an allele and mutate it
    test.genes[random2] = random_gen.rand();

    //evaluate the resultant individual
    test.fitness = test.evaluate(test.genes[0],test.genes[1]);

    //More fit individuals are always accepted and
    //inserted randomly into the next generation
    if(test.fitness < group_of_individuals[random].fitness)
    {
        next_group_of_individuals[random].genes[0] = test.genes[0];
        next_group_of_individuals[random].genes[1] = test.genes[1];
        next_group_of_individuals[random].fitness = test.fitness;
        i--;
    }
    else if(random_gen()<0.7)
    {
        //Less fit individuals are reinserted 70% of the time
        next_group_of_individuals[random].genes[0] = test.genes[0];
        next_group_of_individuals[random].genes[1] = test.genes[1];
        next_group_of_individuals[random].fitness = test.fitness;
    }
}

};

/*
 * This function writes the statistics to a file or terminal
 */
void Population::EchoStatistics(const int iterator,Individual elite)
{
    double avg;
    avg=0;
    for(int i=0;i<POPULATION_SIZE;i++)
    {
        avg += next_group_of_individuals[i].fitness;
    }
}

```



```

    avg = avg/POPULATION_SIZE;
    fstream file;

    cout << iterator << endl;
    file.open ("stats.txt", ios::out | ios::app | ios::binary);
    //    if(iterator==0)
    //    {
    //        file << "Generation" << "          " << "Avg. Fitness" << "    " << "Best Fitness" << endl;
    //    }
    file << iterator << "          " << avg << "    " << elite.fitness << endl;
    file.close();
};

/*
 * This function switches the next generation to the current one.
 */
void Population::GenerationSwitch()
{
    int i,x;
    x = POPULATION_SIZE;
    //max = omp_get_max_threads();
    //omp_set_num_threads(max);
    #pragma omp parallel shared(x) private(i)
    #pragma omp for nowait
    for(i=0;i<x;i++)
    {
        // This will also be affected by my question in "Selection"
        // Copy 'next' into the primary population being 'worked on'
        group_of_individuals[i].genes[0] = next_group_of_individuals[i].genes[0];
        group_of_individuals[i].genes[1] = next_group_of_individuals[i].genes[1];
        group_of_individuals[i].fitness = next_group_of_individuals[i].fitness;

    }

};

// Finds the elite in the CURRENT population
void Population::ElitistSave(Individual *elite)
{
    int k,x;
    x = POPULATION_SIZE;

    double dMin = elite->fitness;
    double dGeneZero = elite->genes[0];
    double dGeneOne = elite->genes[1];

    //Remember, the "best fitness" is the closest to zero!!
    for(k=0;k<x;k++)
    {
        if(group_of_individuals[k].fitness < dMin)
        {

```

```

        //global save
        //elite = cur_population->group_of_individuals[k];

        dMin = group_of_individuals[k].fitness;
        dGeneZero = group_of_individuals[k].genes[0];
        dGeneOne = group_of_individuals[k].genes[1];
    }
}

elite->fitness = dMin;
elite->genes[0] = dGeneZero;
elite->genes[1] = dGeneOne;
};

// Compares what it found initially (in ElitistSave) to the NEXT population
void Population::ElitistCompare(Individual *elite)
{
    double worst,best;
    int i,j,k,worst_tracker,x;
    x = POPULATION_SIZE;

    // Locate the best individual
    for(i=0;i<x;i++)
    {
        // initialize best with the first individual in the array
        best = next_group_of_individuals[0].fitness;

        if(next_group_of_individuals[i].fitness < best)
        {
            best = next_group_of_individuals[i].fitness;
        }
    }

    /* Compare best to the elite
    * If the elite is better, sub in NUM_ELITE times
    * for the worst individual (who must also be located)
    * else, simply drop out -> this results in a
    * computationally minimal ElitistCompare function
    */
    if(best > elite->fitness)
    {
        for(j=0;j<NUM_ELITES;j++)
        {
            // initialize worst with the first individual in the array
            worst = next_group_of_individuals[0].fitness;

            // Find the worst individual
            for(k=0;k<x;k++)
            {
                if(next_group_of_individuals[k].fitness > worst)
                {
                    worst = next_group_of_individuals[k].fitness;
                    worst_tracker = k;
                }
            }
        }
    }
}

```

```

        }
    }

    /* Once the worst individual is replaced by the elite, it will
    * effectually dissappear so I don't have to worry about it being
    * replaced twice upon the next iteration.
    */
    next_group_of_individuals[worst_tracker].genes[0] = elite->genes[0];
    next_group_of_individuals[worst_tracker].genes[1] = elite->genes[1];
    next_group_of_individuals[worst_tracker].fitness = elite->fitness;
}

};

```

**Population.h**

```

/*
 * Population.h
 *
 * Created on: Nov 7, 2009
 * Author: Avery
 */

#ifndef POPULATION_H_
#define POPULATION_H_

//Define for omp
#define OMP_NUM_THREADS 2

#include "Individual.h"
#include "MersenneTwister.h"

//Defines
#define POPULATION_SIZE 200

// Pm in percent
#define PROB_MUT 5

// Pc in percent
#define PROB_CROSS 60

// Number of Elites
#define NUM_ELITES 2

class Population
{
public :
    Population();
    ~Population();

    //average fitness for that generation
    double avg_fitness;

    //where the individual with the best fitness is
    int bst_indi;

    double random_num();
    void Populate();
    void Evaluate(Individual*);
    void Crossover();
    void Selection();
    void Mutation();
    void EchoStatistics(const int i,Individual);
    void GenerationSwitch();
    void ElitistSave(Individual*);
    void ElitistCompare(Individual*);
    void FindAverage();

```

```
        friend class Individual;
        Individual group_of_individuals [POPULATION_SIZE], next_group_of_individuals
[POPULATION_SIZE];
    };

#endif /* POPULATION_H_ */
```

**Individual.cpp**

```

/*
 * Individual.cpp
 *
 * Created on: Nov 6, 2009
 * Author: Avery
 */

/*
 * Individual Class:
 * "Chromosomes" are initialized
 * Fitness is calculated here
 */
#include "Individual.h"
#include "MersenneTwister.h"

double Individual::evaluate(double x, double y){
    //Rosenbrock TF
    double Fitness=0;

    Fitness = pow((1-x),2) + 100*pow(y-pow(x,2),2);
    return Fitness;
};

//Default Constructor, used in array instantiation
Individual::Individual()
{
    genes[0] = 0;
    genes[1] = 0;
    fitness = 0;
};

Individual::~Individual(){};

// Crossing Parent A (this) with Parent B (cur_Individual aka...what's passed in)
// Okay, so this function could be sooo much cooler
// i.e. scan the number of digits in Pa,
// take half the digits and concatenate it to the digits in Pb (for each gene) <- hahaha "single point
crossover!"
Individual Individual::IndividualCrossover(Individual *cur_Individual)
{
    MTRand random_num;
    double random;
    random = random_num.rand();

    Individual child;

    child.genes[0] = random*(cur_Individual->genes[0]) + (1-random)*genes[0];

```



```
child.genes[1] = random*(cur_Individual->genes[1]) + (1-random)*genes[1];  
return child;  
};
```

**Individual.h**

```

/*
 * Individual.h
 *
 * Created on: Nov 6, 2009
 * Author: Avery
 */
#ifndef INDIVIDUAL_H_
#define INDIVIDUAL_H_

#include <stdlib.h>
#include <math.h>
#include <iostream>
using namespace std;

class Individual {
public:
    Individual();
    ~Individual();

    Individual IndividualCrossover(Individual *Pb);
    double evaluate(double,double);
    double fitness;
    double genes [2];
/*
//Overload the equals operator
    virtual Individual& operator=(const Individual& rhs)
    {
        if (this == &rhs);
        {
            return *this;
        }

        //De-allocate all memory associated with the old type
        delete &genes[0];
        delete &genes[1];
        delete &fitness;

        //Assign the old components of the RHS to temp values
        double genes0 = rhs.genes[0];
        double genes1 = rhs.genes[1];
        double new_fitness = rhs.fitness;

        //Re-assign the old components of the RHS to temp values
        genes[0] = genes0;
        genes[1] = genes1;
        fitness = new_fitness;
        //return
        return *this;
    }
}

```

```
};
```

```
#endif /* INDIVIDUAL_H_ */
```

**MersenneTwister.h**

```

// MersenneTwister.h
// Mersenne Twister random number generator -- a C++ class MTRand
// Based on code by Makoto Matsumoto, Takuji Nishimura, and Shawn Cokus
// Richard J. Wagner v1.1 28 September 2009 wagnerr@umich.edu

// The Mersenne Twister is an algorithm for generating random numbers. It
// was designed with consideration of the flaws in various other generators.
// The period,  $2^{19937}-1$ , and the order of equidistribution, 623 dimensions,
// are far greater. The generator is also fast; it avoids multiplication and
// division, and it benefits from caches and pipelines. For more information
// see the inventors' web page at
// http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html

// Reference
// M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-Dimensionally
// Equidistributed Uniform Pseudo-Random Number Generator", ACM Transactions on
// Modeling and Computer Simulation, Vol. 8, No. 1, January 1998, pp 3-30.

// Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
// Copyright (C) 2000 - 2009, Richard J. Wagner
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions
// are met:
//
// 1. Redistributions of source code must retain the above copyright
// notice, this list of conditions and the following disclaimer.
//
// 2. Redistributions in binary form must reproduce the above copyright
// notice, this list of conditions and the following disclaimer in the
// documentation and/or other materials provided with the distribution.
//
// 3. The names of its contributors may not be used to endorse or promote
// products derived from this software without specific prior written
// permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
// IS"
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
// PURPOSE
// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS
// BE
// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
// POSSIBILITY OF SUCH DAMAGE.

```

```

// The original code included the following notice:
//
//   When you use this, send an email to: m-mat@math.sci.hiroshima-u.ac.jp
//   with an appropriate reference to your work.
//
// It would be nice to CC: wagnerr@umich.edu and Cokus@math.washington.edu
// when you write.

#ifndef MERSENNETWISTER_H
#define MERSENNETWISTER_H

// Not thread safe (unless auto-initialization is avoided and each thread has
// its own MTRand object)

#include <iostream>
#include <climits>
#include <cstdio>
#include <ctime>
#include <cmath>

class MTRand {
// Data
public:
    typedef unsigned long uint32; // unsigned integer type, at least 32 bits

    enum { N = 624 }; // length of state vector
    enum { SAVE = N + 1 }; // length of array for save()

protected:
    enum { M = 397 }; // period parameter

    uint32 state[N]; // internal state
    uint32 *pNext; // next value to get from state
    int left; // number of values left before reload needed

// Methods
public:
    MTRand( const uint32 oneSeed ); // initialize with a simple uint32
    MTRand( uint32 *const bigSeed, uint32 const seedLength = N ); // or array
    MTRand(); // auto-initialize with /dev/urandom or time() and clock()
    MTRand( const MTRand& o ); // copy

    // Do NOT use for CRYPTOGRAPHY without securely hashing several returned
    // values together, otherwise the generator state can be learned after
    // reading 624 consecutive values.

    // Access to 32-bit random numbers
    uint32 randInt(); // integer in  $[0, 2^{32}-1]$ 
    uint32 randInt( const uint32 n ); // integer in  $[0, n]$  for  $n < 2^{32}$ 
    double rand(); // real number in  $[0, 1]$ 
    double rand( const double n ); // real number in  $[0, n]$ 
    double randExc(); // real number in  $[0, 1)$ 
    double randExc( const double n ); // real number in  $[0, n)$ 

```

```

double randDblExc();           // real number in (0,1)
double randDblExc( const double n ); // real number in (0,n)
double operator()( );          // same as rand()

// Access to 53-bit random numbers (capacity of IEEE double precision)
double rand53(); // real number in [0,1)

// Access to nonuniform random number distributions
double randNorm( const double mean = 0.0, const double stddev = 1.0 );

// Re-seeding functions with same behavior as initializers
void seed( const uint32 oneSeed );
void seed( uint32 *const bigSeed, const uint32 seedLength = N );
void seed();

// Saving and loading generator state
void save( uint32* saveArray ) const; // to array of size SAVE
void load( uint32 *const loadArray ); // from such array
friend std::ostream& operator<<( std::ostream& os, const MTRand& mtrand );
friend std::istream& operator>>( std::istream& is, MTRand& mtrand );
MTRand& operator=( const MTRand& o );

protected:
void initialize( const uint32 oneSeed );
void reload();
uint32 hiBit( const uint32 u ) const { return u & 0x80000000UL; }
uint32 loBit( const uint32 u ) const { return u & 0x00000001UL; }
uint32 loBits( const uint32 u ) const { return u & 0x7fffffffUL; }
uint32 mixBits( const uint32 u, const uint32 v ) const
    { return hiBit(u) | loBits(v); }
uint32 magic( const uint32 u ) const
    { return loBit(u) ? 0x9908b0dfUL : 0x0UL; }
uint32 twist( const uint32 m, const uint32 s0, const uint32 s1 ) const
    { return m ^ (mixBits(s0,s1)>>1) ^ magic(s1); }
static uint32 hash( time_t t, clock_t c );
};

// Functions are defined in order of usage to assist inlining

inline MTRand::uint32 MTRand::hash( time_t t, clock_t c )
{
    // Get a uint32 from t and c
    // Better than uint32(x) in case x is floating point in [0,1]
    // Based on code by Lawrence Kirby (fred@genesis.demon.co.uk)

    static uint32 differ = 0; // guarantee time-based seeds will change

    uint32 h1 = 0;
    unsigned char *p = (unsigned char *) &t;
    for( size_t i = 0; i < sizeof(t); ++i )
    {
        h1 *= UCHAR_MAX + 2U;
        h1 += p[i];
    }

```



```

    }
    uint32 h2 = 0;
    p = (unsigned char *) &c;
    for( size_t j = 0; j < sizeof(c); ++j )
    {
        h2 *= UCHAR_MAX + 2U;
        h2 += p[j];
    }
    return ( h1 + differ++ ) ^ h2;
}

inline void MTRand::initialize( const uint32 seed )
{
    // Initialize generator state with seed
    // See Knuth TAOCP Vol 2, 3rd Ed, p.106 for multiplier.
    // In previous versions, most significant bits (MSBs) of the seed affect
    // only MSBs of the state array. Modified 9 Jan 2002 by Makoto Matsumoto.
    register uint32 *s = state;
    register uint32 *r = state;
    register int i = 1;
    *s++ = seed & 0xffffffffUL;
    for( ; i < N; ++i )
    {
        *s++ = ( 1812433253UL * ( *r ^ (*r >> 30) ) + i ) & 0xffffffffUL;
        r++;
    }
}

inline void MTRand::reload()
{
    // Generate N new values in state
    // Made clearer and faster by Matthew Bellew (matthew.bellew@home.com)
    static const int MmN = int(M) - int(N); // in case enums are unsigned
    register uint32 *p = state;
    register int i;
    for( i = N - M; i--; ++p )
        *p = twist( p[M], p[0], p[1] );
    for( i = M; --i; ++p )
        *p = twist( p[MmN], p[0], p[1] );
    *p = twist( p[MmN], p[0], state[0] );

    left = N, pNext = state;
}

inline void MTRand::seed( const uint32 oneSeed )
{
    // Seed the generator with a simple uint32
    initialize(oneSeed);
    reload();
}

inline void MTRand::seed( uint32 *const bigSeed, const uint32 seedLength )
{

```

```

// Seed the generator with an array of uint32's
// There are 2^19937-1 possible initial states. This function allows
// all of those to be accessed by providing at least 19937 bits (with a
// default seed length of N = 624 uint32's). Any bits above the lower 32
// in each element are discarded.
// Just call seed() if you want to get array from /dev/urandom
initialize(19650218UL);
register int i = 1;
register uint32 j = 0;
register int k = ( N > seedLength ? N : seedLength );
for( ; k; --k )
{
    state[i] =
        state[i] ^ ( (state[i-1] ^ (state[i-1] >> 30)) * 1664525UL );
    state[i] += ( bigSeed[j] & 0xffffffffUL ) + j;
    state[i] &= 0xffffffffUL;
    ++i; ++j;
    if( i >= N ) { state[0] = state[N-1]; i = 1; }
    if( j >= seedLength ) j = 0;
}
for( k = N - 1; k; --k )
{
    state[i] =
        state[i] ^ ( (state[i-1] ^ (state[i-1] >> 30)) * 1566083941UL );
    state[i] -= i;
    state[i] &= 0xffffffffUL;
    ++i;
    if( i >= N ) { state[0] = state[N-1]; i = 1; }
}
state[0] = 0x80000000UL; // MSB is 1, assuring non-zero initial array
reload();
}

inline void MTRand::seed()
{
    // Seed the generator with an array from /dev/urandom if available
    // Otherwise use a hash of time() and clock() values

    // First try getting an array from /dev/urandom
    FILE* urandom = fopen( "/dev/urandom", "rb" );
    if( urandom )
    {
        uint32 bigSeed[N];
        register uint32 *s = bigSeed;
        register int i = N;
        register bool success = true;
        while( success && i-- )
            success = fread( s++, sizeof(uint32), 1, urandom );
        fclose(urandom);
        if( success ) { seed( bigSeed, N ); return; }
    }

    // Was not successful, so use time() and clock() instead

```

```

        seed( hash( time(NULL), clock() ) );
    }

inline MTRand::MTRand( const uint32 oneSeed )
    { seed(oneSeed); }

inline MTRand::MTRand( uint32 *const bigSeed, const uint32 seedLength )
    { seed(bigSeed,seedLength); }

inline MTRand::MTRand()
    { seed(); }

inline MTRand::MTRand( const MTRand& o )
{
    register const uint32 *t = o.state;
    register uint32 *s = state;
    register int i = N;
    for( ; i--; *s++ = *t++ ) {}
    left = o.left;
    pNext = &state[N-left];
}

inline MTRand::uint32 MTRand::randInt()
{
    // Pull a 32-bit integer from the generator state
    // Every other access function simply transforms the numbers extracted here

    if( left == 0 ) reload();
    --left;

    register uint32 s1;
    s1 = *pNext++;
    s1 ^= (s1 >> 11);
    s1 ^= (s1 << 7) & 0x9d2c5680UL;
    s1 ^= (s1 << 15) & 0xefc60000UL;
    return ( s1 ^ (s1 >> 18) );
}

inline MTRand::uint32 MTRand::randInt( const uint32 n )
{
    // Find which bits are used in n
    // Optimized by Magnus Jonsson (magnus@smartelectronix.com)
    uint32 used = n;
    used |= used >> 1;
    used |= used >> 2;
    used |= used >> 4;
    used |= used >> 8;
    used |= used >> 16;

    // Draw numbers until one is found in [0,n]
    uint32 i;
    do
        i = randInt() & used; // toss unused bits to shorten search

```

```

        while( i > n );
        return i;
    }

    inline double MTRand::rand()
        { return double(randInt()) * (1.0/4294967295.0); }

    inline double MTRand::rand( const double n )
        { return rand() * n; }

    inline double MTRand::randExc()
        { return double(randInt()) * (1.0/4294967296.0); }

    inline double MTRand::randExc( const double n )
        { return randExc() * n; }

    inline double MTRand::randDbExc()
        { return ( double(randInt()) + 0.5 ) * (1.0/4294967296.0); }

    inline double MTRand::randDbExc( const double n )
        { return randDbExc() * n; }

    inline double MTRand::rand53()
    {
        uint32 a = randInt() >> 5, b = randInt() >> 6;
        return ( a * 67108864.0 + b ) * (1.0/9007199254740992.0); // by Isaku Wada
    }

    inline double MTRand::randNorm( const double mean, const double stddev )
    {
        // Return a real number from a normal (Gaussian) distribution with given
        // mean and standard deviation by polar form of Box-Muller transformation
        double x, y, r;
        do
        {
            x = 2.0 * rand() - 1.0;
            y = 2.0 * rand() - 1.0;
            r = x * x + y * y;
        }
        while ( r >= 1.0 || r == 0.0 );
        double s = sqrt( -2.0 * log(r) / r );
        return mean + x * s * stddev;
    }

    inline double MTRand::operator()()
    {
        return rand();
    }

    inline void MTRand::save( uint32* saveArray ) const
    {
        register const uint32 *s = state;
        register uint32 *sa = saveArray;

```

```

        register int i = N;
        for( ; i--; *sa++ = *s++ ) {}
        *sa = left;
    }

inline void MTRand::load( uint32 *const loadArray )
{
    register uint32 *s = state;
    register uint32 *la = loadArray;
    register int i = N;
    for( ; i--; *s++ = *la++ ) {}
    left = *la;
    pNext = &state[N-left];
}

inline std::ostream& operator<<( std::ostream& os, const MTRand& mtrand )
{
    register const MTRand::uint32 *s = mtrand.state;
    register int i = mtrand.N;
    for( ; i--; os << *s++ << "t" ) {}
    return os << mtrand.left;
}

inline std::istream& operator>>( std::istream& is, MTRand& mtrand )
{
    register MTRand::uint32 *s = mtrand.state;
    register int i = mtrand.N;
    for( ; i--; is >> *s++ ) {}
    is >> mtrand.left;
    mtrand.pNext = &mtrand.state[mtrand.N-mtrand.left];
    return is;
}

inline MTRand& MTRand::operator=( const MTRand& o )
{
    if( this == &o ) return (*this);
    register const uint32 *t = o.state;
    register uint32 *s = state;
    register int i = N;
    for( ; i--; *s++ = *t++ ) {}
    left = o.left;
    pNext = &state[N-left];
    return (*this);
}

#endif // MERSENNETWISTER_H

// Change log:
//
// v0.1 - First release on 15 May 2000
//   - Based on code by Makoto Matsumoto, Takuji Nishimura, and Shawn Cokus
//   - Translated from C to C++
//   - Made completely ANSI compliant

```

```

// - Designed convenient interface for initialization, seeding, and
//   obtaining numbers in default or user-defined ranges
// - Added automatic seeding from /dev/urandom or time() and clock()
// - Provided functions for saving and loading generator state
//
// v0.2 - Fixed bug which reloaded generator one step too late
//
// v0.3 - Switched to clearer, faster reload() code from Matthew Bellew
//
// v0.4 - Removed trailing newline in saved generator format to be consistent
//       with output format of built-in types
//
// v0.5 - Improved portability by replacing static const int's with enum's and
//       clarifying return values in seed(); suggested by Eric Heimburg
// - Removed MAXINT constant; use 0xffffffffUL instead
//
// v0.6 - Eliminated seed overflow when uint32 is larger than 32 bits
// - Changed integer [0,n] generator to give better uniformity
//
// v0.7 - Fixed operator precedence ambiguity in reload()
// - Added access for real numbers in (0,1) and (0,n)
//
// v0.8 - Included time.h header to properly support time_t and clock_t
//
// v1.0 - Revised seeding to match 26 Jan 2002 update of Nishimura and Matsumoto
// - Allowed for seeding with arrays of any length
// - Added access for real numbers in [0,1) with 53-bit resolution
// - Added access for real numbers from normal (Gaussian) distributions
// - Increased overall speed by optimizing twist()
// - Doubled speed of integer [0,n] generation
// - Fixed out-of-range number generation on 64-bit machines
// - Improved portability by substituting literal constants for long enum's
// - Changed license from GNU LGPL to BSD
//
// v1.1 - Corrected parameter label in randNorm from "variance" to "stddev"
// - Changed randNorm algorithm from basic to polar form for efficiency
// - Updated includes from deprecated <xxx.h> to standard <xxxx> forms
// - Cleaned declarations and definitions to please Intel compiler
// - Revised twist() operator to work on ones'-complement machines
// - Fixed reload() function to work when N and M are unsigned
// - Added copy constructor and copy operator from Salvador Espana

```

## **CONTACT INFORMATION**

Name: Avery Adam White

Address: Dr. Emily Zechamn  
205-M Wisenbaker Engineering Research Center  
3136 TAMU  
College Station, TX 77843-3136

Email Address: [averyawhite@gmail.com](mailto:averyawhite@gmail.com)

Education: B.S. Computer Engineering, 2010